VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Výpočetní platforma Intel Xeon Phi

Computation platform Intel Xeon Phi

2014 Bc. Pavel Bartošek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Pavel Bartošek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Výpočetní platforma Intel Xeon Phi
Computation Platform Intel Xeon Phi

Zásady pro vypracování:

Cílem práce je prostudovat platformu Intel Xeon Phi, její možnosti, omezení a praktické použití. Dále bude cílem implementovat několik ukázkových příkladů a porovnat její využitelnost s platformou CUDA a OpenCL.

Práce bude obsahovat:

1. Popis platformy Intel Xeon Phi.
2. Porovnání jejích možností s modely OpenCL a CUDA.
3. Implementace několika ukázkových programů.
4. Porovnání efektivity a využitelnosti této platformy s ostatními.

Seznam doporučené odborné literatury:

[1] James Reinders: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism, O'Reilly Media; 1st Ed. edition, 2007, 978-0596514808
[2] James Jeffers, James Reinders: Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufmann; 1 edition , 2013, ISBN-13: 978-0124104143

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014

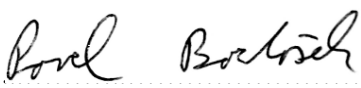
doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 15. 4. 2014



podpis studenta

Chtěl bych poděkovat vedoucímu diplomové práce Ing. Janu Platošovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

Abstrakt

Tato diplomová práce pojednává o možnostech paralelního programování na koprocesoru Intel Xeon Phi. Detailně popisuje architekturu, její vlastnosti, výhody, nevýhody, možnosti použití a celkově problematiku vysoce paralelního programování v dnešní době. Práce dále obsahuje stručný popis ostatních technologií jako OpenCL a CUDA, jejich porovnání s Xeonem Phi na hardwarové úrovni, ale i z pohledu programátora, kde se zaměřuje na kompilátory, programovací jazyky, knihovny, včetně jednotlivých ukázek zdrojového kódu.

Druhá část práce je zaměřena více prakticky, kdy je proveden podrobný test koprocesoru. Je zde vysvětlena problematika kompilace a spouštění výpočetních aplikací v různých režimech na několika konkrétních ukázkách, které jsou pak pod různě nastavenými parametry otestovány. Dále pak ukázka některých užitečných funkcí a vlastností pro usnadnění práce s koprocesorem.

Klíčová slova

Intel Xeon Phi, koprocesor, High Performance Computing, paralelní programování, Offload režim, Nativní režim, Many Integrated Cores, OpenCL, CUDA.

Abstract

This thesis discusses the possibility of parallel programming on Intel Xeon Phi coprocessor. Describes in detail the architecture, its features, advantages, disadvantages, possibilities of use and the overall issue of high parallel programming today. The thesis also contains a brief description of other technologies such as OpenCL and CUDA, comparing them with Xeon Phi at the hardware level, but also from a programmer's perspective, which focuses on compilers, programming languages, libraries, including individual examples of source code.

The second part is more practical when it is made detailed test of coprocessor. There is explain the issue of compiling and running computing applications in a variety of modes for a few specific examples, which are then under a different set parameters tested. In addition, a sample of some useful functions and features to facilitate work with coprocessor.

Keywords

Intel Xeon Phi, coprocesor, High Performance Computing, parallel programming, Offload mode, Native mode, Many Integrated Cores, OpenCL, CUDA.

Seznam použitých symbolů a zkratek

MIC - Many Integrated Cores

HPC - High Performance Computing

VPU - Vector Processing Unit

ALU - Arithmetic Logic Unit

EMU - Extended Math unit

SP - Single Precision

DP - Double Precision

TD - Tag Directory

SIMD - Single Instruction Multiple Data

MIMD - Multiple Instruction Multiple Data

SSE - Streaming SIMD Extensions

AVX - Advanced Vector Extensions Instructions

Cluster - Spojení více počítačů, které považujeme za jediný systém

AoS - Array of Structures

SoA - Structure of Arrays

MPI - Message Passing Interface

TBB – Threading Building Blocks

OpenCL – Open Computing Language

OpenMP – Open Multi-Processing

CUDA – Compute Unified Device Architecture

FLOP/s - Floating point operations per second

FMA – Fused Multiply and Add

MPI - Message Passing Interface

TDP - Thermal Design Power

SSH – Secured Shell

Seznam obrázků

Seznam výpisků zdrojového kódu

Seznam tabulek

Obsah

1 Úvod

V dnešní době se s termínem paralelního programování setkáváme mnohem častěji, než tomu bylo třeba před 10 lety. Není to až tak překvapující vzhledem k masivnímu rozvoji hardwaru pro paralelní výpočty v posledních letech, zejména pak u HPC. Nejedná se již pouze o samotné procesory, které tyto výpočty zpracovávaly, ale pomalu se do popředí dostávají speciální stand-alone výpočetní karty určené speciálně pro ty nejnáročnější výpočty. Tyto karty disponují oproti běžným procesorům několikanásobně vyšším počtem jader, pamětí a celkově úplně odlišnou koncepcí zaměřenou na vysoký výpočetní výkon a nízkou spotřebu energie.

Nevýhoda těchto výpočetních karet spočívá v jejich použití pro nevhodné aplikace. Jejich největší potenciál tkví ve více vláknových aplikacích, kdy je využito řádově desítek až stovek vláken. Dalšími nevýhodami jsou například pořizovací cena, mnohdy nedostačující sdílená paměť nebo nízká propustnost sběrnice.

Tato diplomová práce je zaměřena na úplnou novinku na poli vysoce paralelního programování. Je jí přídavná karta do PCI-E slotu Intel Xeon Phi vycházející z modifikované architektury Pentium, která je pojmenovaná jako MIC. Její největší předností oproti konkurenční NVIDIA Tesla je možnost použití široké škály aplikací implementovaných v různých programovacích jazycích (např. OpenCL, OpenMP, Cilk/Cilk Plus, Fortran, C++) a využití mnoha knihoven a doplňků. Celá práce pak pojednává o architektuře, vlastnostech a použití karty, její výhody a nevýhody a také praktické ukázky implementace. Samozřejmě nechybí ani detailní srovnání s jinými technologiemi jako jsou OpenCL a CUDA, ze kterého je jasně patrné, která technologie je na tom nejlépe a v čem ostatní zaostávají.

2 Popis platformy Intel Xeon Phi

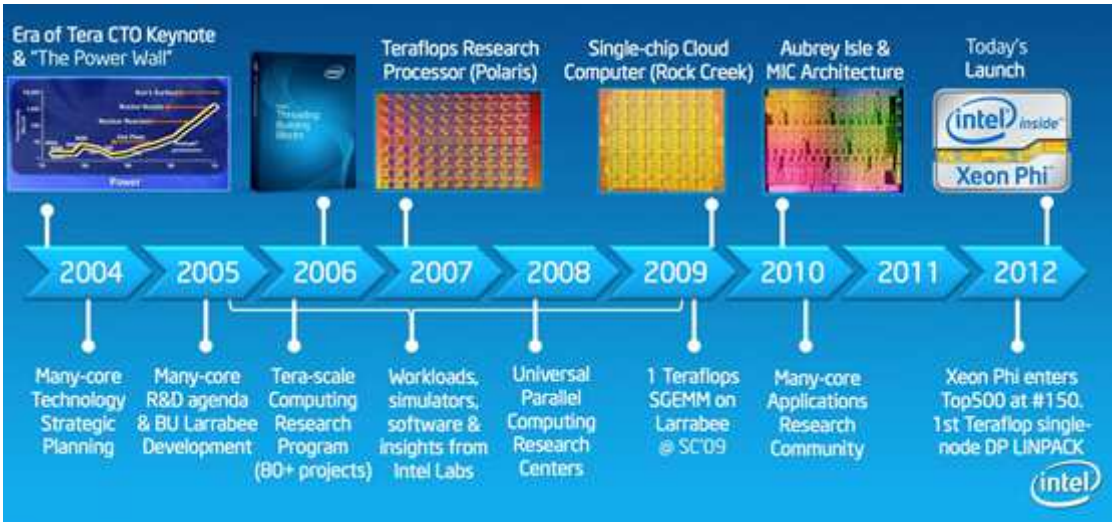
Na začátku všeho, je vhodné se seznámit se samotnou architekturou Intel Xeonu Phi na základní úrovni. Přiblížíme si různé technické novinky, kterými karta disponuje a nenajdeme je ani u konkurence, vnitřní propojení, logické a výpočetní jednotky, práci s cache pamětmi, způsoby napájení a úspory energie, ale také samotnou historii a prvopočátky vývoje. V neposlední řadě také specifikace jednotlivých karet pro různé segmenty trhu a důležitá fakta o těchto paralelních výpočetních kartách.

2.1 Historie vývoje paralelních čipů Intel

Společnost Intel se poměrně dlouhou dobu snažila vyvinout výpočetní čip s vysokým paralelním výkonem. I přes neúspěch s platformou Larrabee se jí to ale nakonec podařilo. S příchodem první akcelerační karty pro 3D výpočty se v počítačích začaly samostatně rozvíjet dvě hlavní výpočetní části. Na jedné straně byl tradiční procesor, který má jedno či více jader na vysoké frekvenci, je univerzální a schopný zpracovat různé druhy úloh. Na druhé straně grafický čip, který má mnoho menších výpočetních celků pro specializované druhy výpočtů při zpracování obrazu.

V začátcích 3D akcelerace to byli oba současní konkurenti AMD a Nvidia, kteří začali udávat směr vývoje grafických čipů. Postupným vývojem se ale začaly grafické čipy zaměřovat na větší a větší rozsah výpočtů a architektura se nakonec stala velmi univerzální i pro obecné výpočty, které s grafikou nemají vůbec nic společného.

Intel teoreticky odstartoval výzkum mnohojádrových výpočetních čipů v roce 2004 v rámci programu Tera-Scale. Kvůli absenci výkonné grafické karty se Intel snažil vyrobit grafickou kartu s kódovým označením Larrabaee, která by byla dostatečně výkonná a efektivní při srovnání s modely od AMD a Nvidie. Vývoj Larrabee byl ale neúspěšný, protože ani s těmi nejdražšími dostupnými inženýry se nepodařilo vyvinout produkt požadovaných vlastností. Mnoholeté zkušenosti AMD a Nvidie byly v tomto ohledu neocenitelné. Architektura grafických čipů se ale postupem času stávala univerzálnější a Intel se už nemusel soustředit na složité konstrukce pro 3D výpočty, extrémně složité ovladače a kompatibilitu. Intel následně vyvinul ještě několik univerzálních paralelních výpočetních čipů, které již určovaly ten správný směr, jež společnost zachránil před velkým neúspěchem v této oblasti. Avšak tyto čipy byly oproti konkurenci slabší a energeticky náročnější. Jednalo se o čipy Aubrey Isle a Rock Creek, které nebyly masově vyráběny. Obrázek 1 ukazuje vývojovou mapu paralelních čipů společnosti až po současnost.



Obrázek 1: Vývojová mapa paralelních čipů společnosti Intel [7]

2.2 Intel Xeon Phi hardwarová architektura

Přídomek Phi je anglický přepis řeckého písmene fí, má evokovat souvislost s vědou a zejména pak zlatým řezem, který φ nejčastěji symbolizuje. Označení Xeon bylo zvoleno proto, aby koprocesory MIC (Many Integrated Cores) byly součástí již existující rodiny produktů pro servery a výpočetní stanice.

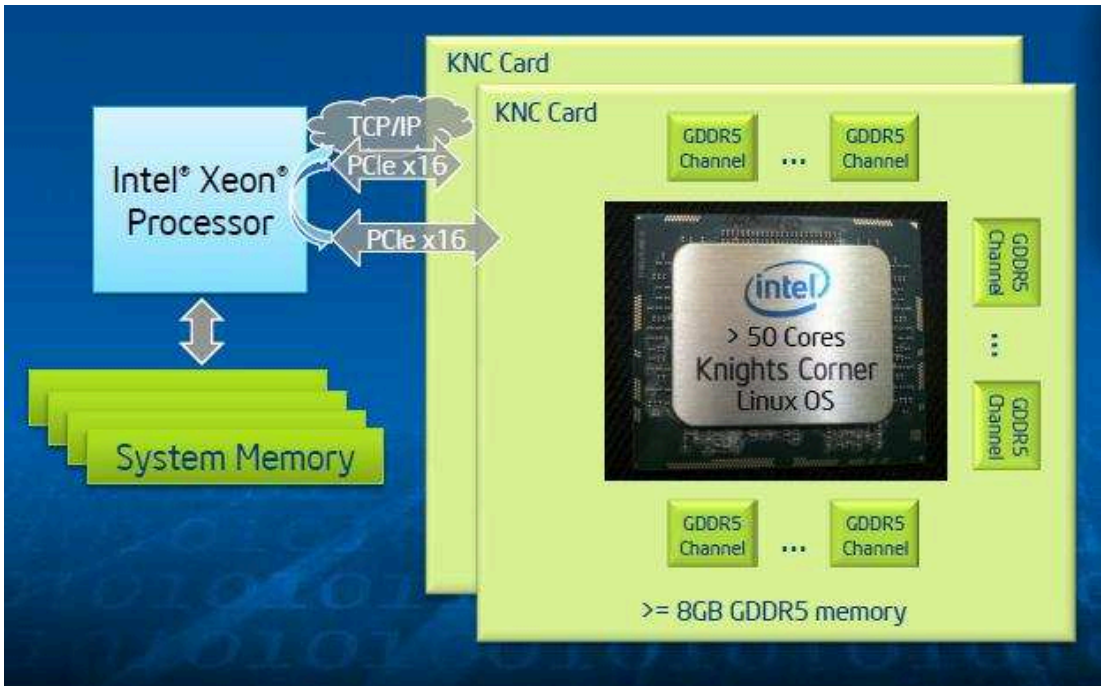
Projekt mnohojádrových výpočetních koprocesorů rodiny Knights, též nazývaných MIC, se původně vyvinul z nikdy nevydaného grafického jádra Larrabee. Karty Knights Ferry byly určené pouze pro vývojáře jako testovací platforma. První generace Intel Xeon Phi s kódovým označením Knights Corner je již určena pro komerční sféru.

Oproti výrobcům AMD nebo Nvidie, kteří u svých univerzálních grafických čipů používají specializovanou architekturu, se Intel rozhodl kvůli zpětné kompatibilitě a snazšímu rozšíření použít instrukční sadu x86. MIC architektura v sobě spojuje mnoho jader, CPU do jediného čipu a je určena pro vysoce paralelní zátěž (HPC) v různých oblastech, jako jsou fyzikální výpočty, chemie, biologie nebo finanční služby. V dnešní době jsou takové výpočty provozovány jako paralelní aplikace na velkých výpočetních stanicích.

Intel MIC architektura se zaměřuje na dosažení vysoké propustnost v prostředí clusterů. Základním atributem mikroarchitektury je, že je postavena tak, aby poskytovala univerzální programovací prostředí. Intel Xeon Phi koprocesory založené na architektuře Intel MIC fungují plně na operačním systému Linux. Na čipu (respektive v paměti karty) totiž běží vlastní operační systém založený na Linuxu. Ten je zaveden hostitelským počítačem při inicializaci. Poté s ním lze komunikovat pomocí protokolu TCP/IP, jako by se jednalo o vzdálený počítač.

Rozdělování a spouštění úloh tedy není řešeno prostřednictvím ovladače jako u GPU, ale klasicky, spouštěním programu na hostovaném operačním systému. V serveru s osazenými Xeony Phi tak běží cluster vnořených počítačů. K inicializaci sice potřebují pomoc hostujícího systému, jinak ale běží samostatně. Tento model znamená mnohem snazší programování ve srovnání s architekturou výpočetních GPU. Karty podporují x86 paměťový model a IEEE 754 s plovoucí řádovou čárkou. Jsou schopny spouštět aplikace napsané ve standardních programovacích jazycích jako Fortran, C a C++. Koprocesor je podporován bohatým vývojovým prostředím, které zahrnuje kompilátory, četné knihovny mezi nimi například knihovny pro práci s vlákny nebo matematické knihovny, výkonnostní charakteristiku a ladící nástroje.

Intel Xeon Phi koprocesor je připojen k hostitelskému procesoru prostřednictvím PCI Express (PCIe) sběrnice (viz Obrázek 2). Vzhledem k tomu, Intel Xeon Phi koprocesor běží na operačním systému Linux, může být virtualizované TCP/IP spojení provedeno po sběrnici PCIe. To umožňuje uživateli přístup ke koprocesoru jako síťovému uzlu. Proto se může každý uživatel připojit ke koprocesoru prostřednictvím zabezpečeného prostředí a přímo spouštět jednotlivé úlohy nebo sérii úloh. Koprocesor podporuje heterogenní aplikace, kde se část aplikace spustí na hostiteli (procesoru), zatímco jinou část vykonává koprocesor.

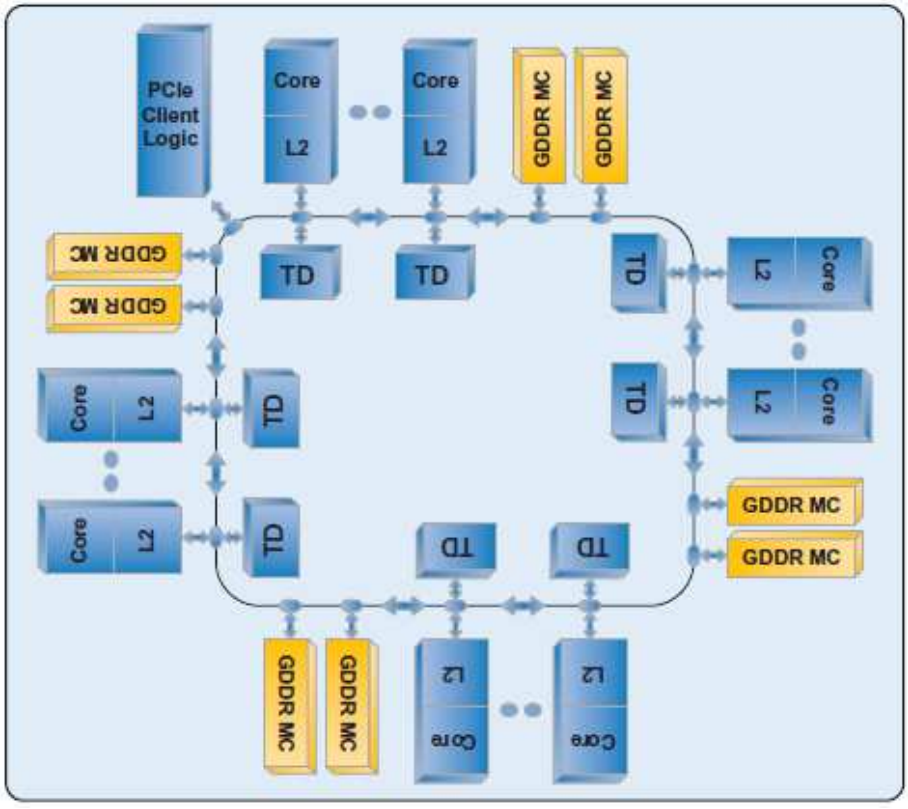


Obrázek 2: První generace Intel Xeon Phi s kódovým označením Knights Corner [7]

Velkou výhodou také je, že větší počet Intel Xeon Phi koprocesorů může být instalováno do jednoho hostitelského systému. V rámci jednoho systému, mohou koprocesory komunikovat

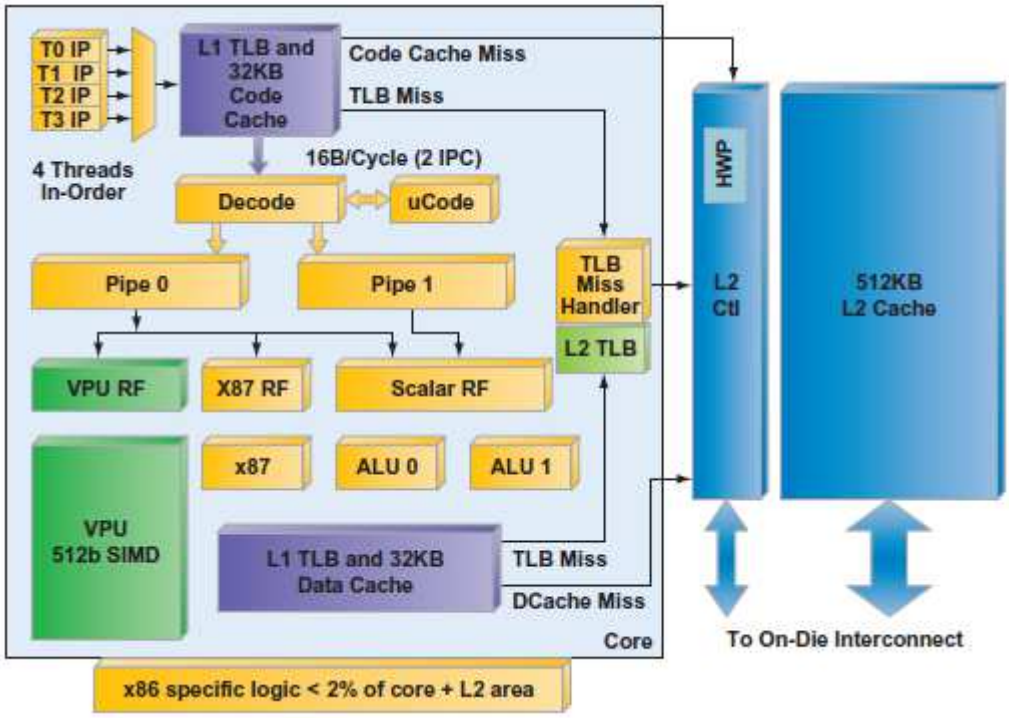
spolu navzájem přes PCIe peer-to-peer propojení bez jakéhokoli zásahu hostitele. Stejně tak je možné, aby koprocesory komunikovaly prostřednictvím síťové karty, jako je Ethernet nebo InfiniBand, bez jakéhokoli zásahu hostitele.

Intel Xeon Phi koprocesor je primárně složen z procesorových jader, cache, paměťových řadičů, PCIe logického klienta a vysoce propustnou, obousměrnou propojovací prstencovou sběrnici (viz Obrázek 3). Každé jádro má vlastní 512 KB L2 cache. L1 cache má kapacitu 32 KB pro data a stejné množství pro instrukce. Paměťové řadiče a PCIe logický klient poskytují přímý přístup do paměti GDDR5 koprocesoru a sběrnice PCIe. Všechny tyto komponenty jsou spojeny dohromady v kruhu.



Obrázek 3: Mikroarchitektura [1]

Každé jádro (Obrázek 4) v koprocesoru Intel Xeon Phi je navrženo jako energeticky úsporné a zároveň poskytuje vysoký výkon pro vysoce paralelní zatížení. Bližší pohled ukazuje, že jádro používá 2 krátké in-order pipeline, které jsou schopny využívat až 4 vlákna. Jedna z nich dokáže zpracovat pouze běžné instrukce v ALU, druhá navíc podporuje instrukce x87 a vektorové instrukce pomocí jednotek FPU, respektive VPU. ALU i FPU jsou poměrně jednoduché a mají zejména podpůrnou funkci.



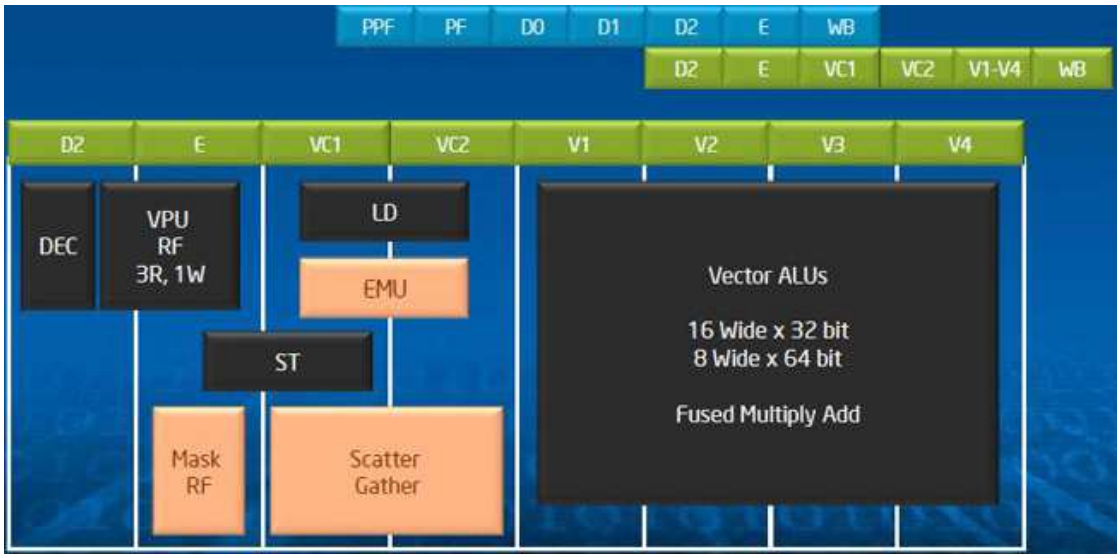
Obrázek 4: Jádro koprocesoru Intel Xeon Phi [1]

2.2.1 Vektorová jednotka

Důležitou součástí jádra koprocesoru je Vektorová jednotka (Vector Processing Unit na Obrázku 5). Jednotka VPU nepodporuje existující instrukce SIMD, jako jsou SSE či AVX, místo toho používá speciální sadu, která byla dříve známa jako „Larrabee New Instructions“. Ta mimo jiné obsahuje operace scatter/gather (shromažďování a rozdělování) a FMA (fused multiply-add), která dokáže s třemi čísly naráz provést ekvivalent násobení a součtu. Právě VPU a instrukce SIMD tvoří hlavní výpočetní sílu Xeonu Phi. Šířka najednou zpracovávaného vektoru je celých 512 bitů (pro srovnání – u MMX je šířka 64 bitů, u SSE 128 a u AVX 256 bitů). Tak může provádět VPU 16 operací s jednoduchou přesností (SP) nebo 8 operací s dvojitou přesností (DP) v jednom cyklu. VPU také podporuje instrukce násobení - sčítání (FMA), a proto je možné provést 32 SP nebo 16 DP operací s plovoucí čárkou během jednoho cyklu. Poskytuje také podporu pro celá čísla.

Vektorové jednotky jsou velmi úsporné z hlediska pracovního vytížení HPC. Do jediné operace lze nasměrovat velké množství výpočtů a nevznikají zbytečné náklady na energie spojené s ukládáním, dekódováním nebo rušením instrukcí. Nicméně, několik zlepšení bylo nutné provést pro tak široké SIMD instrukce. Například maskovací registr byl přidán do VPU, aby bylo možné lépe předpovědět vykonávání instrukcí. To pomohlo při vectorizingu krátké podmíněné větve a tím se docílilo zlepšení celkové efektivity softwarového řetězení. VPU také

podporuje shromažďování a rozdělování instrukcí, které nevyžadují přístup přímo do paměti. Také pro kódy ojedinělých nebo nepravidelně přístupových vzorů, vektorové rozdělování a shromažďování instrukcí, pomoc při udržování vektorových kódů.



Obrázek 5: Vektorová jednotka uvnitř jádra koprocesoru [7]

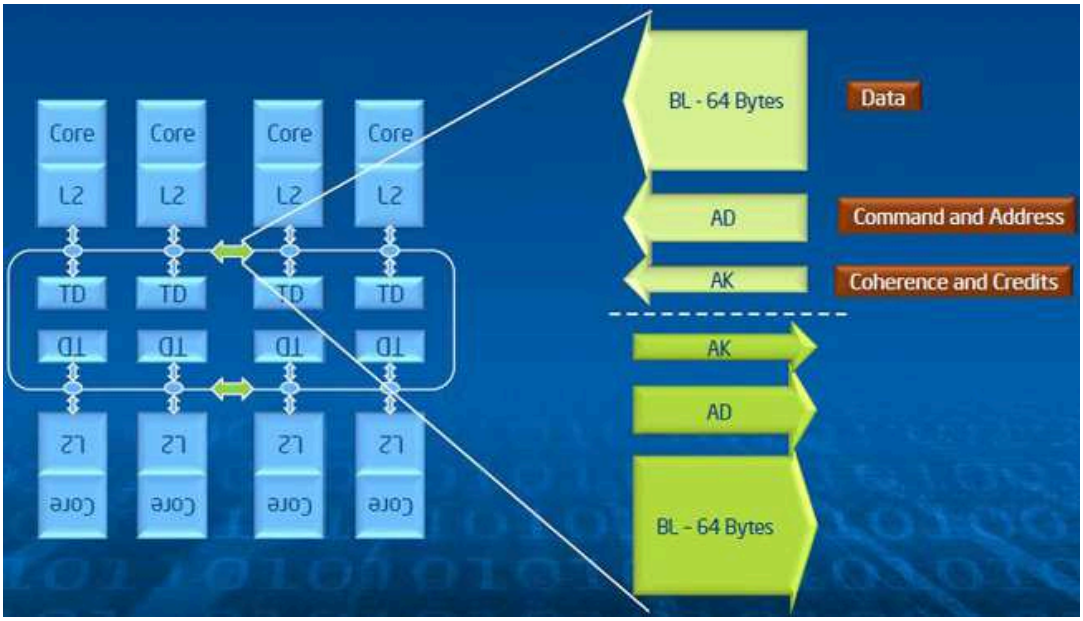
VPU má také rozšířenou matematickou jednotku (EMU), ve které může spustit transcendentální operace, jako je odmocnina a logaritmus, což umožňuje tyto operace, které mají být provedeny ve vektorovém režimu s vysokou propustností. EMU funguje na základě výpočtu polynomu aproximace těchto funkcí.

Koprocesor nepoužívá technologii „out of order execution“, nýbrž instrukce vykonává jednu po druhé v původním pořadí. Takové architektury (mezi jinými například Atom) ovšem trpí ztrátami výkonu, neboť velmi často nemohou zaměstnat své výpočetní jednotky. K tomu dojde tehdy, pokud nelze vykonat dvě instrukce paralelně kvůli závislosti a také v případě, že procesor nemá k dispozici potřebná data a musí na ně čekat.

Intel Xeon Phi tento problém řeší podobně jako Atom, a sice variací na technologii HyperThreading. Jedno jádro zpracovává paralelně čtyři vlákna, takže pokud jedno z vláken jádro nevytíží, lze volnou kapacitu přidělit dalšímu, aniž by došlo k prostoji. Je třeba si uvědomit, že daní za toto řešení je značně snížený výkon na jedno vlákno (neboť čtyři vlákna se dělí o jádro, které zpracovává nanejvýš dvě instrukce naráz). Výměnou za to však vzroste celkový výkon jádra při paralelním zpracování.

2.2.2 Vnitřní propojení

Vnitřní propojení (Obrázek 6) je vytvořeno obousměrnou prstencovou sběrnicí, na které se nachází i logika komunikující s rozhraním PCI Express potažmo s hostitelským strojem. V každém směru se skládá ze tří samostatných menších linek. První, největší a nejdražší z nich je datová linka, která je 64B široká a podporuje vysoké požadavky na propustnost vzhledem k velkému počtu jader. Adresní linka je mnohem menší a slouží k odeslání read/write příkazů a paměťových adres. Poslední nejmenší linka je potvrzovací, která řídí tok a soudržnost zprávy.

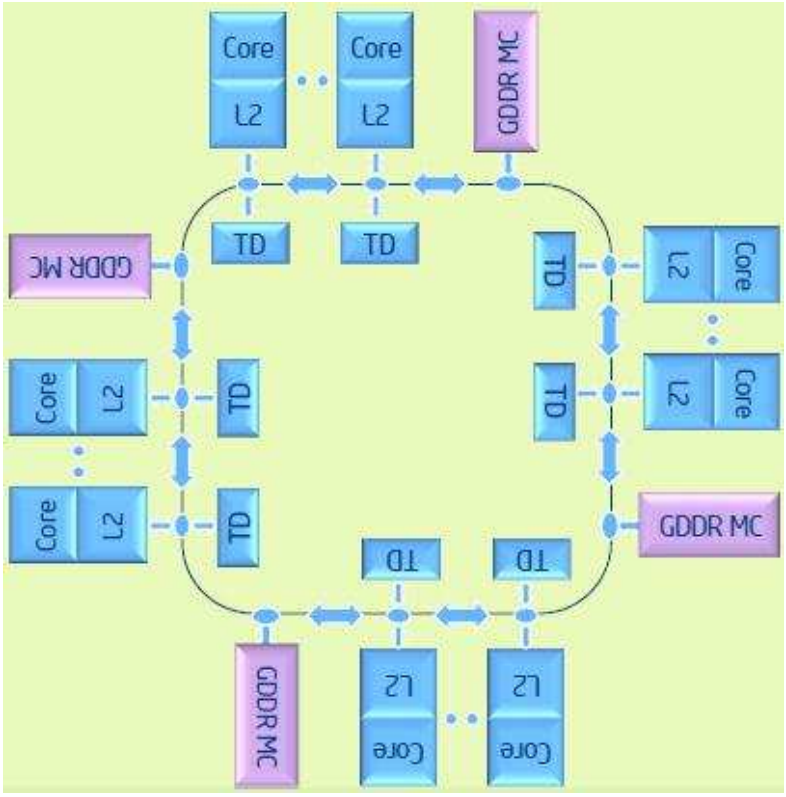


Obrázek 6: Schéma vnitřního propojení [7]

Když jádro přistupuje ke své L2 cache a nezná adresu, je požadavek odeslán přes adresní linku do adresáře tagů (TD – tag directory). Z něj lze zjistit, zda se určitá data nacházejí v konkrétní paměti cache. Jádra tedy tyto záznamy používají pro prohledávání cache svých blízkých i vzdálených sousedů. Pokud se totiž data nacházejí v některé z pamětí cache, není třeba je zdlouhavě načítat z hlavní paměti. Adresy paměti jsou rovnoměrně distribuovány do jednotlivých TD na sběrnici, aby byl zajištěn hladký provoz sběrnice. V případě, že požadovaný datový blok se nachází v L2 cache jiného jádra, je odeslán požadavek předávání do této L2 cache přes adresní linku a datový blok je následně předán přes datovou linku. Pokud požadovaná data nebyla nalezena v žádné cache, je adresa paměti poslána z TD do paměťového řadiče, kde jsou pak dále vyhledávána v hlavní paměti.

Obrázek 7 ukazuje distribuci paměťových řadičů na obousměrné sběrnici. Řadiče paměti jsou souměrně prokládané kolem prstenu. K dispozici je all-to-all mapování z TD do

paměťových řadičů. Adresy jsou rovnoměrně rozděleny mezi paměťové řadiče, což eliminuje aktivní oblasti a poskytuje jednotný přístup, který je důležitý pro dobrou odezvu sběrnice.



Obrázek 7: Schéma vnitřního propojení přístupu do paměti [7]

Při přístupu do paměti, když L2 cache nemá potřebnou informaci pro jádro, si jádro generuje požadavky na adresy přes adresní linku a dotazy ukládá do TD. Pokud data nebyla nalezena v TD, jádro generuje jinou adresní žádost pro data. Jakmile paměťový řadič načte blok dat z paměti, je vrácen zpět do jádra po datové lince. Takto v průběhu procesu jsou jeden datový blok a dva dotazy na adresu (a podle protokolu, dvě potvrzovací zprávy) přenášeny linkami. Vzhledem k tomu, že datové linky jsou velmi drahé a jsou navrženy tak, aby mohly podporovat požadovanou datovou propustnost, bylo třeba zdvojit počet levnějších adresových a potvrzovacích linek, neboť velké množství přenášených paketů (komunikuje mezi sebou téměř stovka klientů) by jinak sběrnici přetěžovalo. Ty musí odpovídat zvýšenému požadavku na propustnost v důsledku vyššího počtu žádostí na těchto linkách. Tím bylo zajištěno nejlepšího výkonu pro všech více jak 50 jader a výsledný výkon karty stoupl o 40%.

2.2.3 Ostatní konstrukční vlastnosti

Mezi další vylepšení architektury začleněné do koprocesoru Intel Xeon Phi patří 64-vstupní Překladový Lookside Buffer (TLB), simultánní load/store datové cache a 512 KB L2

cache. Koprocesor realizuje 16-ti streamový hardwarový Prefetcher ke zlepšení tzv. cachehits (úspěšně nalezených dat v cache) a poskytuje větší propustnost.

Intel MIC architektura využívá hůře L1 a L2 cache ve srovnání s GPU architekturou. Intel Xeon Phi koprocesor realizuje špičkový, vysokorychlostní paměťový subsystém. Každé jádro je vybaveno instrukční 32 KB L1 cache, datovou 32KB L1 cache a jednotnou 512 KB L2 cache. L1 a L2 cache poskytují celkovou propustnost, která je přibližně 15 a 7x rychlejší ve srovnání s celkovou propustností paměti. Proto efektivnější využití cache je klíčem k dosažení špičkového výkonu koprocesoru. Kromě zlepšení propustnosti, jsou cache také energeticky účinnější pro dodávání dat do jádra, než paměti. Spotřebovaná energie na bajt přenesených dat z paměti je mnohonásobně vyšší než u L1 a L2 cache. V době eskalujícího výpočetního výkonu hraje použití cache klíčovou roli v dosažení skutečného výkonu při dodržení přísných energetických omezení.

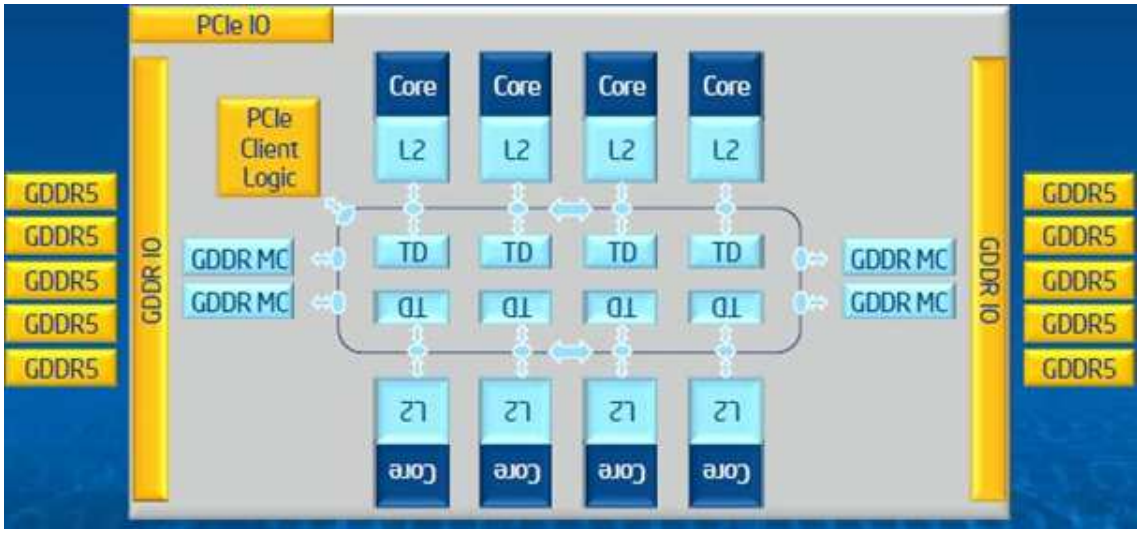
Dalším užitečnou funkcí je využívání šablon. Ty jsou běžné například ve fyzikálních simulacích a jedná se o klasické příklady zatížení, které vykazují velké zvýšení výkonu prostřednictvím efektivního využívání cache. Jestliže pracovní zátěž není naprogramována tak, aby byla v cache zablokována, bude snížena propustnost paměti. Blokování cache slibuje značný nárůst výkonu vzhledem k větší propustnosti a energetické účinnosti cache ve srovnání s pamětí. Blokování cache zlepšuje výkon tím, že blokuje fyzickou strukturu nebo fyzický systém tak, že zablokované data se dostanou dobře do L1 nebo L2 cache konkrétních jader. Například v každém časovém kroku může stejné jádro zpracovávat data, které se stále nachází v L2 cache z předchozího kroku, a proto nemusí být načtena z paměti. Tím je dosaženo výrazného zvýšení výkonu. Soudržnost pamětí cache dále umožňuje šablonám operace automatického načítání aktualizovaných údajů z nejbližších sousedních bloků, které jsou umístěny v L2 cache jiných jader. Tak šablony jasně demonstrují přínos efektivního využití pamětí cache a soudržnost v zatížení HPC.

2.2.4 Power Management

Intel Xeon Phi koprocesory nejsou vhodné pro všechny typy úloh. V některých případech je výhodné spouštět úlohy pouze na hostitelském procesoru. V takových situacích, kdy koprocesor není používán, je nutné, aby přešel do úsporného režimu. Z důvodu úspory energie, jakmile se zastaví všechna čtyři vlákna v jádru, je toto jádro odpojeno od hodinového signálu (stav C1). Pokud je takto odpojeno po nastavený čas, je odpojeno i od energie (jejich L2 cache však zůstává k dispozici ostatním jádrům). Tento stav se nazývá C6. Po probuzení je ovšem třeba je znovu inicializovat. Takto může být v každém místě libovolný počet jader vypnut nebo

zapnut. Navíc když jsou všechny jádra odpojeny od energie a není zjištěna žádná aktivita, je odpojen hodinový signál od prstencové sběrnice i všech jejích klientů (jader i s L2 cache a TD, paměťových řadičů) vyjma připojení k PCIe (viz Obrázek 8).

V tuto chvíli, může hostitelský ovladač dát koprocesor do hlubšího spánku, kde všechny jádra jsou odpojeny od energie. Paměti GDDR jsou v obnovovacím režimu a logika PCIe je ve stavu čekání na probuzení. Pouze GDDR IO pamětí spotřebovávají jen velmi málo energie. Tento nejúspornější stav nazýváme Package C6. Opětovné probuzení potom vyžaduje částečné resetování čipu. Tyto techniky řízení napájení pomáhají šetřit energii, což je velmi důležité, pokud by tyto koprocesory byly využívány například v datových centrech.



Obrázek 8: Jádra odpojeny od napájení a TD, L2, řadiče GDDR odpojeny od signálu [7]

2.2.5 Specifikace modelů karet Xeon Phi

První generace koprocesorů Intel Xeon Phi postaveném na čipu Knights Corner vyráběná 22 nm technologií je velmi rozmanitá. Můžeme si vybrat podle toho, jak velký výpočetní výkon očekáváme nebo jakým způsobem je řešeno chlazení karty. Z výkonnostního pohledu jsou karty děleny do 3 kategorií (viz Obrázek 9).

Nejslabší řada 3100 nabízí 2 verze karet 3120P a 3120A. Obě mají 57 jader na 1,1 GHz (což dává výkon 1003 GFLOPS v dvojité přesnosti), 6 GB paměti GDDR5 na taktu 5 GHz (propustnost 240 GB/s) a 28,5 MB cache, která se odvíjí od počtu jader. Tyto vykazují 300 W náročnost TDP. Obě verze mají identické parametry.

Střední třída označovaná jako 5100 a nabízí 2 verze karet. Starší 5110P a novější 5120D. Model 5120D je o něco výkonnější než 5110P, ale také energeticky náročnější (spotřebuje 245 místo 225 W). Zatímco čip je taktován stejně a má identický počet jader (60 kusů na 1,1 GHz,

celkový výkon 1053 GFLOPs), zrychlily paměti, a to o 10 % na 5,5 GHz, kdy se zvýšila propustnost na 352 GB/s, přičemž kapacita je 8 GB. Řada 5100 je určena tam, kde se požaduje vyvážený poměr výkonu a spotřeby.

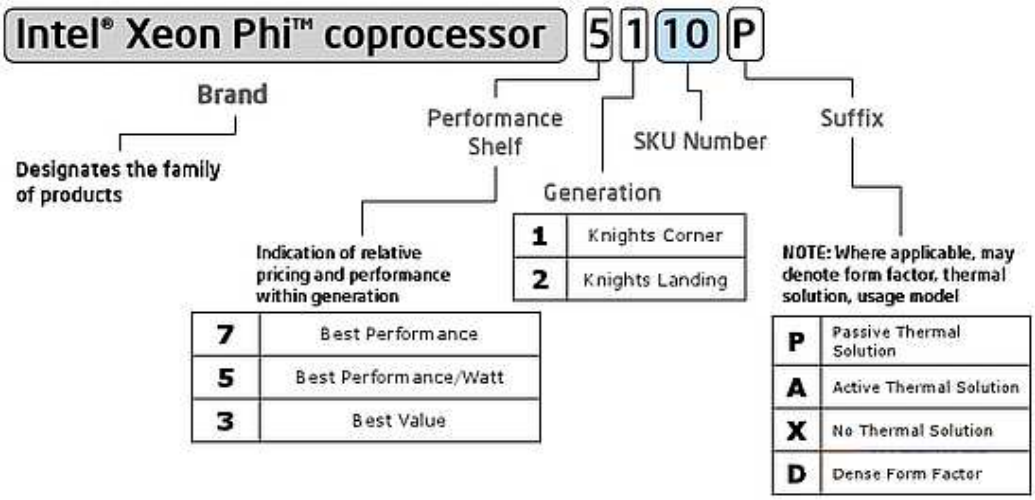
Nevyšší řada 7100 nabízí rovněž 2 karty 7120P a 7120X, které pak tvoří výkonnostní vrchol nabídky. Jsou ještě rychlejší než SE10P/X (tyto karty oficiálně v nabídce nejsou, Intel je ale dodává vybraným zákazníkům), byť mají stejné 300 W TDP. Tyto karty mají aktivních 61 jader z celkových 62 a o poznání vyšší frekvenci 1,25 GHz. Díky tomu se dostávají na teoretický výkon 1220 GFLOPs. Paměti jsou opět taktovány na 5,5 GHz s propustností 352 GB/s a opět je osazeno 8 GB. Obě verze mají identické parametry a jsou určeny pro nevyšší segment trhu.

Processor Brand Name	Codename	Process	SKU #	Form Factor, Thermal	Board TDP (Watts)	Max # of Cores	Clock Speed (GHz)	Peak Double Precision (GFLOP)	GDDR5 Memory Speeds (GT/s)	Peak Memory BW	Memory Capacity (GB)	Total Cache (MB)	Production Si Stepping	Turbo Enabled
Intel® Xeon Phi™ Coprocessor	Knights Corner	22nm	SE10P	PCIe Card, Passively Cooled	300	61	1.1	1073.6	5.5	352	8	30.5	B	N
			SE10X	PCIe Card, No Thermal Solution	300	61	1.1	1073.6	5.5	352	8	30.5	B	N
			5110P	PCIe Card, Passively Cooled	225	60	1.053	1011	5.0	320	8	30	B, C	N
			7120P	PCIe Card, Passively Cooled	300	61	1.25	1220	5.5	352	8	30.5	C	TBD
			7120X	PCIe Card, No Thermal Solution	300	61	1.25	1220	5.5	352	8	30.5	C	TBD
			5120D	Dense Form, No Thermal Solution	245	60	1.053	1011	5.5	352	8	30	C	TBD
			3120P	PCIe Card, Passively Cooled	300	57	1.1	1003	5.0	240	6	28.5	C	TBD
			3120A	PCIe Card, Actively Cooled	300	57	1.1	1003	5.0	240	6	28.5	C	TBD

Obrázek 9: Kompletní specifikace první generace koprocesorů Intel Xeon Phi [4]

Písmeno na konci každého modelu, neboli sufix, označuje typ chlazení karty. Verze A znamená aktivní chladič, verze P pasivní. Písmeno D v názvu značí „Dense Form Factor," což znamená, že karty jsou kompaktnější, aby se s nimi dalo dosáhnout vyšší hustoty integrace. Dodávají se bez chladiče, takže odvod tepla si budou muset obstarat sami odběratelé například vodním chlazením. Verze X chladič neobsahuje a je tomu stejně jako u verze D.

Další zajímavostí je použitá revize čipů u jednotlivých modelů. Na obrázku 9 je možné vidět ještě jednu věc. Zatímco první vyrobené karty byly založeny na revizi (steppingu) B, karty z druhé vlny již budou dodávány s čipy revize C. Novější karty, by tak mohly mít lepší poměr spotřeby a výkonu, což umožnilo vydání výkonnějších modelů řady 7100. Další v pořadí druhá generace ponese kódové označení Knights Landing, které značí příští, čtvrtou generaci čipů Larrabee. Až tato generace přijde, budou její členové odlišeni dvojkou na druhém místě v modelovém čísle (viz Obrázek 10) [7].



Obrázek 10: Značení modelů Intel Xeon Phi [4]

2.2.6 Fakta o první generaci karet Xeonu Phi

- Koprocesor vyžaduje nejméně jeden systémový procesor
- Na kartě běží skutečný SMP on-chip x86 Linux
- Výrobní proces je 22 nm od Intelu s 3-D třícestnými tranzistory
- Podpora vývojového nástroje Intel Parallel Studio XE 2013 a jiných podpůrných nástrojů
- Více než 50 jader (liší se u různých modelových řad a budoucích generací)
- Každá karta má svou IP adresu, připojení probíhá přes terminál
- In-order jádra podporující 64 bitové x86 instrukce s unikátně širokým možnostmi SIMD
- 4 hardwarová vlákna pro každé jádro (více než 200 hardwarových vláken na jedné kartě) primárně využité ke snížení latencí. V praxi použití nejméně 2 vláken na jádro k dosažení nejlepších výsledků.
- Všechny jádra vzájemně propojena vysokorychlostní obousměrnou sběrnicí
- Takt jader je 1 GHz nebo více
- Soudržnost cache napříč celým koprocesorem
- Každé jádro má svou lokální 512 KB L2 cache s vysokorychlostním přístupem do všech ostatních
- Cache poskytují vysoce efektivní využití napájení a zároveň nabízí vysokou propustnost pamětí
- Speciální instrukce kromě x86
- Instrukce Intel MMX, SSE, AVX

- Vysoký výkon pro výpočet odmocniny a operace s exponentem
- Scatter/gather streamování k dosažení vyšší efektivity propustnosti paměti
- Paměťový kontroler podporující více než 8 GB GDDR5 paměti (liší se u různých modelových řad a budoucích generací)
- Logika PCIe připojení přímo na čipu
- Velké možnosti při řízení spotřeby
- Možnosti monitorování výkonnosti pro nástroje jako je Intel VTune Amplifier XE 2013

3 Seznámení s platformami CUDA a OpenCL

V této rozsáhlé kapitole jak již název napovídá, se seznámíme s dvěma platformami, které jsou pro Xeon Phi největší konkurencí. Jedná se o platformu CUDA od společnosti NVIDIA a o platformu OpenCL od konsorcia Khronos Group. Obě si detailně přiblížíme od doby samotného vzniku, jejich základních rysů a vysvětlíme si, jaké jsou jejich klady a zda existují nějaké zápory. Dále pak z čeho se skládají, jaké modely, jazyk a frameworky využívají a nastíníme výhled do budoucnosti jejich vývoje. Jak později budeme moci zjistit, aplikace vytvořené pod těmito platformami je možné spouštět na Xeonu Phi bez větších obtíží.

3.1 Platforma OpenCL

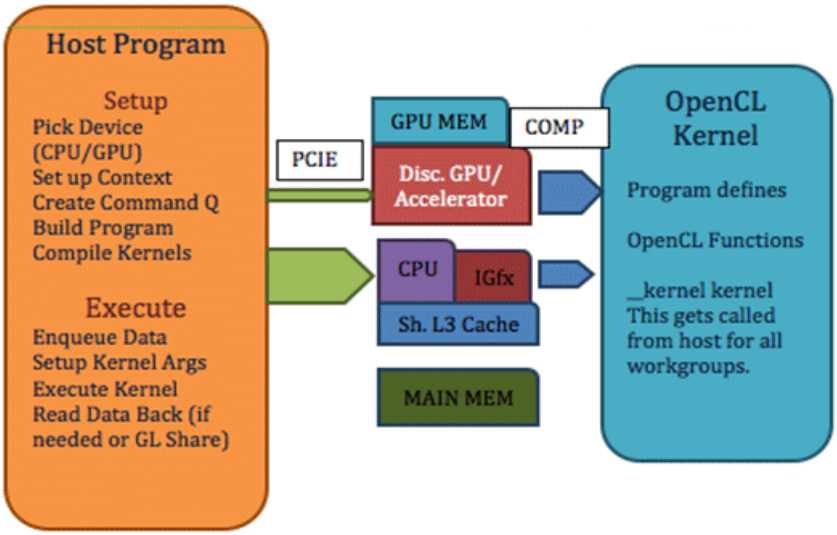
OpenCL (Open Computing Language) je průmyslový standard pro paralelní programování heterogenních počítačových systémů. Pro mnohé moderní hardwarové architektury se paralelismus stává jedinou cestou k vyšším výkonům. Tváří v tvář fyzikálním limitům, nejčastěji v podobě teplotních omezení, se různé platformy vydaly jednotnou cestou navyšování počtů exekučních jednotek umožňujících paralelní zpracování, čím dál většího množství dat. Zároveň s tím dochází k rychlému nárůstu výkonu a schopnosti i těch nejmenších mobilních zařízení. Množství nosných architektur pro náročné aplikace tak roste poměrně rychlým tempem. Dosavadní prostředí, která se tuto situaci pokoušejí řešit, trpí závažnými nedostatky. Mnohá proprietární řešení jsou totiž vázaná na konkrétní hardware (CUDA) či software (DirectCompute). Další nástroje sice umožňují jistý stupeň přenositelnosti avšak za cenu znatelně pomalejšího běhu aplikací na ně postavených.

Tento stav má za cíl řešit průmyslový standard OpenCL, jehož prvotní návrh lze vystopovat až k firmě Apple, která dosud drží práva k názvu OpenCL. V polovině roku 2008 přešel vývoj tohoto návrhu do rukou průmyslového konsorcia Khronos, které za tímto účelem vytvořilo pracovní skupinu Khronos Compute Working Group [9]. Tato skupina zahrnovala členské zastoupení nejvýznamnějších firem v oboru jako AMD, IBM, Intel a nVidia. Aktuální verze OpenCL je 2.0, která je určena k dalšímu zjednodušení cross-platformního programování, a zároveň umožňuje bohatou škálu algoritmů a programovacích vzorů pro celkové zrychlení výpočtů. Jako základ pro tyto možnosti, OpenCL 2.0 definuje rozšířený exekuční model a podskupinu C11 a C++11 paměťový model, synchronizaci a atomové operace.

OpenCL definuje abstraktní hardwarové zařízení a k němu ovládací softwarové rozhraní (viz Obrázek 11), pomocí kterého aplikace přistupují ke konkrétním výpočetním možnostem různých hardwarových platforem. Jednoduchost modelu abstraktního zařízení usnadňuje jeho

implementaci na široké škále existujících i plánovaných hardwarových platforem. Tyto platformy zahrnují klasické procesory, grafické procesory, signální procesory, některé novější mobilní čipy, procesory typu Cell a další.

Ani softwarové rozhraní standardu není závislé na softwarové platformě, což znamená, že nepotřebuje ke svému chodu žádný konkrétní operační systém. Výrobci grafických čipů již zahrnuli implementaci OpenCL do grafických ovladačů nabízených pro nejrozšířenější operační systémy či hlavní distribuce Linuxu. Podpora samozřejmě nechybí ani v produktech Apple a je dokonce dostupná i v některých virtuálních strojích jako je VMware.



Obrázek 11: Nejvyšší úroveň hardwarové a softwarové architektury OpenCL [9]

Standard OpenCL obsahuje několik hlavních části:

- Abstraktní modely určující požadované vlastnosti a chování zařízení OpenCL.
- OpenCL Framework jehož součásti je definice OpenCL API.
- Specifikaci programovacího jazyka, který je využíván pro programování zařízení OpenCL (OpenCL C).

3.1.1 Modely OpenCL

Modely OpenCL jsou abstraktním vyjádřením vlastností a chování platforem a zařízení, jenž odpovídají standardu OpenCL. Jsou to:

- Model platformy
- Exekuční model
- Paměťový model
- Programovací model

3.1.1.1 Model platformy

Model platformy definuje heterogenní paralelní stroj jako počítačový systém schopný nabízet služby OpenCL. Tento heterogenní stroj obsahuje hostitelský systém (dále jen hostitel) a jedno či více zařízení OpenCL (dále jen zařízení), kterými hostitel disponuje. Model platformy navíc předpokládá, že se zařízení skládá z výpočetních jednotek, které jsou dále dělené do procesních elementů.

3.1.1.2 Exekuční model

Exekuční model slouží k řízení běhu softwarového systému využívající OpenCL probíhá na dvou úrovních heterogenního paralelního stroje. Klasická aplikační část (dále jen aplikace) je vykonávaná v rámci hostitele. Tato aplikace odpovídá kromě jiného za komunikaci mezi hostitelem a zařízeními stejně jako za spuštění a koordinaci výpočtů na těchto zařízeních. Samotné zařízení zpracovává tu část aplikace, jež byla vyjádřena jazykem OpenCL C (dále jen program). Program má formu jednoho či více výpočtových vláken, které jsou zpracovávané v rámci procesních elementů daného zařízení. Tato vlákna jsou instancemi funkčního objektu, který se nazývá kernel. Při spuštění výpočtu specifikuje aplikace jedno až třírozměrný indexový prostor, celkový počet instancí kernelu a velikost skupin, do kterých se budou tyto instance sdružovat. Z těchto informací OpenCL následně určí počet skupin a každé instanci kernelu přiřadí globální index, lokální index a skupinový index. Lokální respektive globální index identifikuje instanci kernelu v rámci skupiny respektive v rámci všech instancí. Skupinový index je identifikátor skupiny, do něhož instance kernelu patři.

V rámci exekučního modelu si aplikace vytváří kontexty. V kontextu jsou zahrnuté informace o zařízeních, množina kernelů a programech OpenCL v němž jsou kernely uložené, paměťových objektech, které kernely budou zpracovávat a další údaje. Pro ovládání zařízení, jejich synchronizaci, přesun dat a spouštění kernelů slouží příkazové fronty. Tyto příkazové fronty jsou schopné zpracovávat příkazy „in order“ (jak jdou za sebou) nebo „out of order“ (nezávislé na pořadí). Každý příkaz při zařazení do fronty generuje událost. Vzhledem k tomu, že většinu příkazů je možné vykonávat asynchronně lze tyto události využívat pro sledování stavu příkazů a jejích vzájemnou synchronizaci.

3.1.1.3 Paměťový model

Paměťový model slouží k definici paměťové hierarchie, která zahrnuje různé paměťové oblasti zařízení. Specifikuje typy paměti, druhy přístupu a její alokaci, kdo je za tyto operace odpovědný a jak.

- Globální paměť je oblast paměti viditelná všem instancím kernelu.
- Konstantní paměť je oblast globální paměti, do nichž instance kernelu nemá právo zápisu.
- Lokální paměť je oblast paměti viditelná pouze instancím kernelu ve skupině.
- Privátní paměť je oblast paměti viditelná pouze v rámci instance kernelu.

3.1.1.4 Programovací model

Z exekučního modelu vyplývá, že OpenCL podporuje úlohově paralelní a datově paralelní programovací modely (čí kombinaci obou). Standard OpenCL se soustřeďuje hlavně na datově paralelní programovací model. Tento model definuje výpočet jako souběh instancí kernelu zpracovávajících datové složky vstupní datové struktury. V rámci indexového prostoru exekučního modelu jsou tyto instance a způsob jejích mapování na tyto datové složky definovaný jednoznačně. V nejjednodušším případě připadá jedna instance kernelu na jednu datovou složku, neplatí to však vždy. Úlohově paralelní programovací model (multitasking) umožňuje souběžně spouštět několik instancí různých kernelů. V tomto modelu není však možné vyžadovat těsný souběh několika instancí stejného kernelu. To je hlavní rozdíl oproti datově paralelnímu programovacímu modelu, kdy se v jednu chvíli spouští mnoho instancí jednoho kernelu, které mohou spolu úzce komunikovat.

3.1.2 OpenCL Framework

OpenCL Framework poskytuje aplikacím možnost využívat hostitelský systém a jeho zařízení v souladu s modely OpenCL jako heterogenní paralelní stroj. Framework obsahuje následující komponenty:

- Programovací aplikační rozhraní OpenCL API, jež umožňuje práci se systémem OpenCL.
- Kompilátor jazyka OpenCL C, který překládá programy psané v jazyce OpenCL C do konkrétního strojového kódu dané hardwarové platformy.

3.1.3 Jazyk OpenCL C

Programovací jazyk OpenCL C je založen na normě „ISO/IEC 9899:1999 - Specifikace jazyka C“ (dále C99). Oproti C99 se OpenCL C liší množstvím rozšíření, která zahrnují:

- Vektorové datové typy
- Datové typy a funkce podporující práci s obrázky a jejích filtrování
- Kvalifikátory adresního prostoru
- Kvalifikátory přístupových práv

- Kernelové funkce

Na druhou stranu OpenCL C zavádí několik omezení:

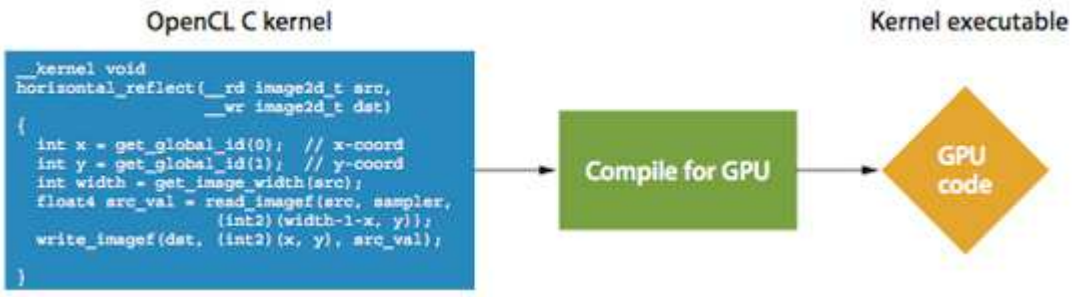
- Ukazatele na funkce, pole proměnné délky a bitová pole jsou zakázána
- Velká většina hlavičkových souborů standardní knihovny jazyka C je nedostupná
- Rekurzivní funkce nejsou povolené
- Kernelové funkce nesmějí deklarovat argumenty typu ukazatel na ukazatel ani nic vracet
- Zápisy na pole číselných typů menších než 32 bitů jsou zakázané

Následující ukázka kódu ukazuje implementaci sečtení dvou polí v jazyce OpenCL C. Struktura je u takovýchto aplikací většinou stejná. Nachází se zde hlavní funkce Main, ve které dochází k alokaci paměti na zařízení, deklaraci proměnných nebo výběru a inicializaci zařízení. Důležitou částí je funkce kernelu. Jedná se o část programu prováděnou na zařízení (viz Obrázek 12) v tomto případě jednoduchý součet dvou polí (viz Ukázka 1). Tento kernel je volán z funkce Main, která je vykonávána na hostovi. Odtud jsou poslány data do zařízení, kde se vykoná funkce kernel a výsledné zpracované data jsou poté vrácena hostovi. Tento proces je vždy stejný a je to i jeden z důvodů možnosti spouštění OpenCL kódu na platformě CUDA [10].

```
#include <iostream>
#include <CL/cl.hpp>
int main() {
    // vytvoření vyrovnávací paměti zařízení
    cl::Buffer buffer_A(context,CL_MEM_READ_WRITE,sizeof(int)*10);
    cl::Buffer buffer_B(context,CL_MEM_READ_WRITE,sizeof(int)*10);
    cl::Buffer buffer_C(context,CL_MEM_READ_WRITE,sizeof(int)*10);

    int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int B[] = {0, 1, 2, 0, 1, 2, 0, 1, 2, 0};

    //vytvoření fronty pro plnění příkazů
    cl::CommandQueue queue(context,default_device);
    //vyhledá všechny platformy
        std::vector<cl::Platform> all_platforms;
        cl::Platform::get(&all_platforms);
        if(all_platforms.size()==0){
        std::cout<<" No platforms found. Check OpenCL installation!\n";
        exit(1);
    }
    cl::Platform default_platform=all_platforms[0];
    std::cout << "Using platform: "<<default_platform.getInfo<CL_PLATFORM_NAME>()<<"\n";
    //vybere defaultní platformu
    std::vector<cl::Device> all_devices;
    default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);

    if(all_devices.size()==0) {
        std::cout<<" No devices found. Check OpenCL installation!\n";
        exit(1);
    }
    cl::Device default_device=all_devices[0];
    std::cout<< "Using device: "<<default_device.getInfo<CL_DEVICE_NAME>()<<"\n";
    cl::Context context({default_device});
    cl::Program::Sources sources;

    // kernel spočítá C=A+B
    std::string kernel_code=
```

```
    " void kernel simple_add(global const int* A, global const int* B, global int* C) { "
    " C[get_global_id(0)]=A[get_global_id(0)]+B[get_global_id(0)]; "
    " } ";
    sources.push_back({kernel_code.c_str(),kernel_code.length()});

    cl::Program program(context,sources);
    if(program.build({default_device})!=CL_SUCCESS) {
        std::cout<<" Error building:
"<<program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(default_device)<<"\n";
        exit(1);
    }

    //zapíše pole A a B do zařízení
    queue.enqueueWriteBuffer(buffer_A,CL_TRUE,0,sizeof(int)*10,A);
    queue.enqueueWriteBuffer(buffer_B,CL_TRUE,0,sizeof(int)*10,B);

    //spustí kernel
    cl::KernelFunctor
    simple_add(cl::Kernel(program,"simple_add"),queue,cl::NullRange,cl::NDRange(10),
    cl::NullRange);
    simple_add(buffer_A,buffer_B,buffer_C);

    int C[10];
    //přečte výsledke ze zařízení a uloží do pole C
    queue.enqueueReadBuffer(buffer_C,CL_TRUE,0,sizeof(int)*10,C);
    //zobrazí výsledek z pole C
    std::cout<<" result: \n";
    for(int i=0;i<10;i++) {
        std::cout<<C[i]<<" ";
    }

    return 0;
}
```

Ukázka 1: Součet dvou polí v OpenCL



Obrázek 12: Schéma vykonání kernelu OpenCL na GPU [12]

3.1.4 Požadavky pro práci s OpenCL

Podmínkou vývoje aplikací pod OpenCL je mít nainstalován některý z vývojových kitů ATi Stream SDK nebo nVidia OpenCL SDK nejnovější verze. Mezi podporované operační systémy patří Windows XP/Vista/7/8, Ubuntu 9.10 a vyšší a Red Hat 5.3 a vyšší. Mezi podporované kompilery pak patří Microsoft Visual Studio (MSVS) 2005 a vyšší nebo GNU Compiler Collection (GCC) 4.1.2 a vyšší.

Z hlediska podporovaného hardwaru se jedná o grafické karty nVidia GeForce, nVidia GeForce Mobile, Quadro FX nebo NVidia Tesla, dále pak grafické karty ATI Radeon HD, ATi

FireGL, ATi FireMV, ATi FirePro, AMD FireStream, ATI Mobility Radeon HD. Z řad procesorů se jedná o procesory AMD nebo Intel x86 s minimální instrukční sadou SSE 3.x a vyšší. Pro samotný vývoj je velmi důležité mít vždy nejnovější verzi SDK, protože nejnovější grafické karty a procesory nejsou podporovány u starších verzí SDK jako je OpenCL 1.0 nebo 1.1.

Kromě výchozího profilu standardu je k dispozici též profil pro mobilní zařízení, která jsou schopná podporovat modely architektury OpenCL, ale nedisponují dostatečným výkonem pro zajištění plného rozsahu funkčnosti. V tomto profilu jsou některé části standardu nepovinné (podpora 3D obrazu) či odstraněné úplně (64bitové číselné typy).

3.1.5 Výhody a nevýhody platformy OpenCL

Hlavní a největší výhodou je nezávislost na hardwaru, na kterém jsou výpočty prováděny a fakt, že OpenCL je multiplatformní. Nejsme tedy omezeni konkrétním hardwarem jako je tomu například u technologie CUDA. S tím souvisí relativně nízké pořizovací náklady v porovnání s technologií Xeon Phi. I v dnešní době výkonných osmijádrových procesorů je však výhodnější z hlediska výkonu tyto výpočty provádět na grafických kartách, které většinu času zahálejí, a jejich potenciál není využit. OpenCL je otevřený standard. To znamená, že jej může použít kdokoliv a má tedy otevřený přístup k této normě stejně tak ke všem dokumentacím, tutoriálům, open source nástrojům a jiným pomůckám. Při použití OpenCL výpočetního modelu také dochází k velkému šetření energie kvůli ušetřenému výpočetnímu času. Kompilace kódu je možné provádět online i offline, což je oproti platformě CUDA nesporně výhoda. OpenCL standard je v dnešní době podporován již mezi několika desítkami výrobců hardwaru a polovodičových součástí, což vede k ještě masivnějšímu rozšíření tohoto standardu i do budoucna. Z hlediska programovacího jazyka je OpenCL C99 založen na jazyku C, což umožnilo vřelé přijetí mezi programátory. Nový jazyk by se tak snadno neuchytil a tento standard by jistě nebyl natolik rozšířený. OpenCL kernely mohou být volány z jazyků jako C, C++, Java, Python, JavaScript, Haskell, Perl, Ruby a seznam stále roste. To umožňuje přenositelnost a opětovné využití stávajících OpenCL kernelu a vede k flexibilní podpoře pro výpočty GPGPU v celé řadě vývojových prostředí. OpenCL používá přesnější speciální funkce ve výchozím nastavení (samozřejmostí je použití nativních funkcí). Většina počítačů mají OpenCL kompatibilní grafickou kartu. Pokud nemáte GPU ve vašem systému (nepravděpodobný scénář), jsou k dispozici ovladače umožňující běžící OpenCL programy přímo na CPU. Instruktážní videa a návody jsou široce dostupné a aplikační poznámky a

příklady programů jsou poskytovány mnoha GPU dodavateli. Je snadné a levné začít experimentovat s OpenCL.

Nevýhod tohoto standardu není mnoho. Mezi největší patří fakt, že u nového hardwaru není zaručeno, že bude kompatibilní s OpenCL, dokud tento hardware nebude zahrnut v nejnovějších verzích SDK. OpenCL nemá přístup k několika hardwarovým instrukcím a algoritmům oproti technologii CUD, proto musí využívat pomalejší metody přístupu (textury, cache size selection). Dalším omezením je pak přenositelnost binárního distribučního formuláře, které klade OpenCL standard v nevýhodu v porovnání s jinými normami. Přidělování paměti a práce s ní by mohlo probíhat rychleji, to samé platí o přístupu do paměti.

3.1.6 Budoucnost OpenCL

Ačkoli je standard v současné době zmrazen, mají Khronos Group a jednotliví vývojáři volnou ruku při navrhování rozšiřující funkcionality. Dosavadní rozšíření například specifikují spolupráci OpenCL API s grafickými API (OpenGL, DirectX) nebo odstraňují některá omezení jazyka OpenCL C.

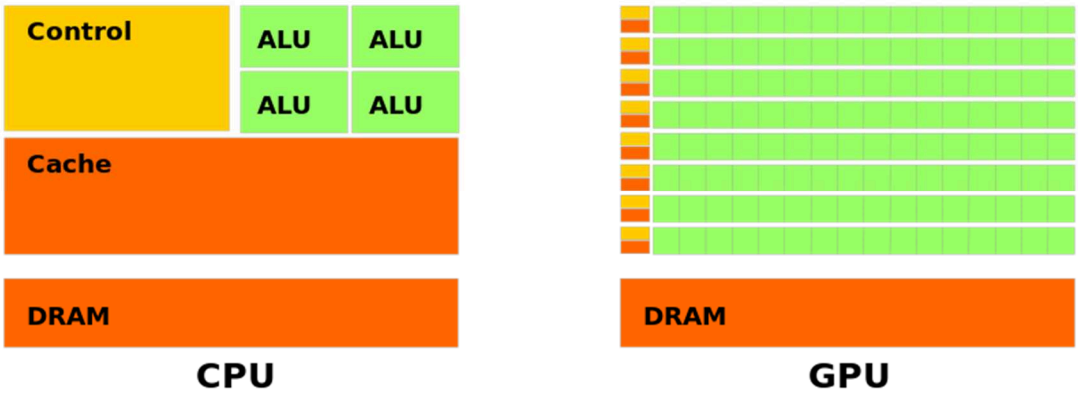
Příznivců této technologie je v dnešní době mnoho a je téměř nemožné určit, která technologie je lepší nebo horší, každá nabízí něco jiného. Lidé, kteří vyvíjejí pod OpenCL tak činí hlavně z důvodu multiplatformnosti, že nemusí psát přesně specifický kód pro NVIDA CUDA nebo ATI Stream, aby plně využili těchto architektur. Například kódy vhodné pro NVIDIA CUDA je možné využít na konkurenční ATI Stream jen s 20% výkonem, což je docela zásadní. OpenCL technologie byla dříve pomalejší, ale s postupem moderní implementace a nových verzí SDK je stejně rychlá, jako CUDA. To však platí pouze v případě, že je implementace OpenCL provedena přesně pro danou architekturu, pak není o nic horší než CUDA. Vzhledem k tomu, že hlavním rysem OpenCL je přenositelnost (přes abstraktní paměti a exekuční model), programátor není schopen přímo používat GPU-specifické technologie (např. inline PTX), pokud není ochoten vzdát se přímé přenositelnosti.

3.2 Platforma CUDA

Společnost NVIDIA je známá pro své čipové sady základních desek, stejně jako pro vynikající grafické procesory, které se staly populární jako základ pro grafické karty. Ve snaze o maximální rychlost se NVIDIA GPU vyvinuly daleko za hranicemi jednotlivých procesorů. Moderní NVIDIA GPU nejsou jednotlivé procesory, ale paralelní superpočítače na čipu, který se skládá z velmi mnoho rychlých procesorů. Současné NVIDIA GPU s rozmezím 16 až 1536 stream procesorů na kartu, poskytující neuvěřitelně silný výpočetní výkon i šířku pásma.

CUDA (Compute Unified Device Architecture) je hardwarová a softwarová architektura, která umožňuje na GPU spouštět programy napsané v jazycích C/C++, FORTRAN nebo programy postavené na technologiích OpenCL, DirectCompute a jiných [11]. Tato architektura je dostupná pouze na grafických akcelerátorech společnosti NVIDIA, která ji vyvinula. Konkurenční technologie společnosti AMD se nazývá ATI Stream. Obě společnosti jsou také členy Khronos Group, která zajišťuje vývoj OpenCL.

Technologii představila společnost NVIDIA v roce 2006. První verze SDK 1.0 byla určena pro karty NVIDIA Tesla založené na architektuře G80, později bylo vydáno SDK ve verzi 1.1, které přidala podporu pro karty série GeForce verze 8. Se správným ovladačem grafické karty přibyla podpora pro překrývání paměťových přenosů výpočtem a podpora pro více GPU akcelerátorů. S pozdějšími verzemi přibývaly další vylepšení jako podpora pro emulovaný výpočet v double precision, podpora pro C++ šablony v rámci kernelu, nativní podpora pro double precision výpočty, podpora pro ukazatele na funkce a podpora rekurze, Vylepšeny jsou také profilovací nástroje a debuggery pro CUDA / OpenCL, unifikace paměťových prostorů a masivní podpora MultiGPU. Nejnovější stabilní verze je CUDA SDK 5.5, kde přibyla podpora ARM architektury a celkově došlo ke stabilizaci a zrychlení výpočtů.

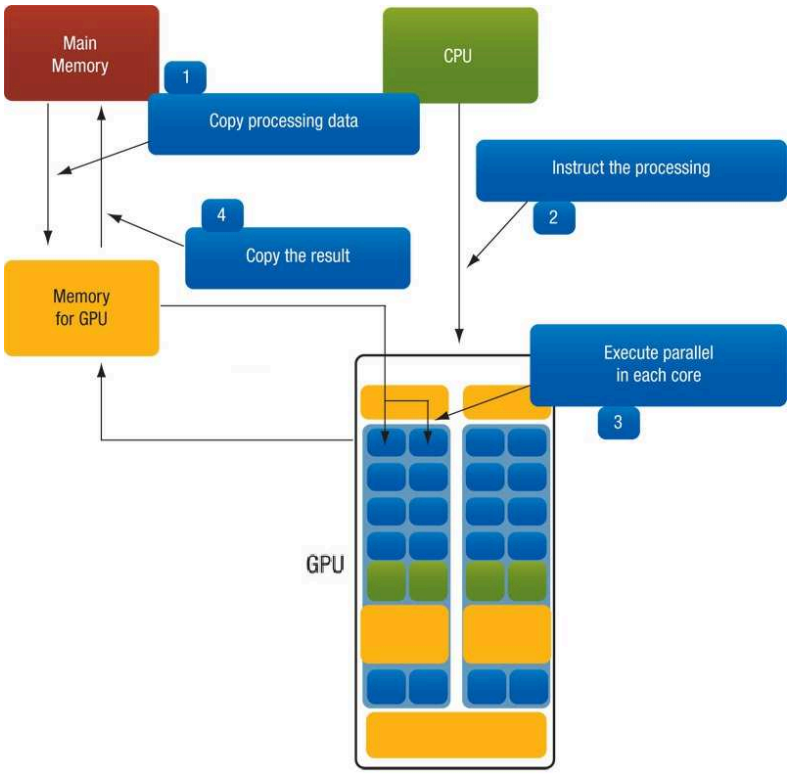


Obrázek 13: Architektura CPU vs GPU [12]

Drtivou většinu plochy čipu grafického akcelerátoru zabírá velké množství relativně jednoduchých skalárních procesorů (viz Obrázek 13), které jsou organizovány do větších celků zvaných streaming multiprocesory. Vzhledem k tomu, že se jedná o SIMT architekturu, řízení jednotek a plánování instrukcí je jednoduché a spolu s velmi malou vyrovnávací pamětí zabírá malé procento plochy GPU čipu. To má bohužel za následek omezené predikce skoků a časté zdržení výpadky cache (některé typy pamětí dokonce nejsou opatřeny cache). Poslední významnou částí, která je rozměrově velice podobná CPU, je RAM řadič. Obecně se multiprocesor skládá z několika (dnes až z 1536) stream procesorů, pole registrů, sdílené

paměti, několika load/store jednotek a Special Function Unit - jednotky pro výpočet složitějších funkcí jako sin, cos, ln.

Typický průběh GPGPU výpočtu probíhá tak, že nejprve je vyhrazena část paměti na GPU pro výpočet, poté následuje přesun dat z hlavní paměti RAM do paměti grafického akcelerátoru. Následně je spuštěn výpočet na grafické kartě a po skončení je výsledek z paměti grafické karty přesunut zpět do hlavní RAM paměti (viz Obrázek 14).



Obrázek 14: Typický průběh GPGPU výpočtu [12]

3.2.1 Modely CUDA

3.2.1.1 Programovací model

Programovací model CUDA definuje výpočet jako souběh instancí kernelu. CUDA aplikace je složena z částí, které běží buď na hostu (CPU) nebo na CUDA zařízení (GPU). Části aplikace běžící na zařízení jsou spouštěny hostem zavoláním kernelu, což je funkce, která je prováděna každým spuštěným vláknem (*thread*). Definujeme 3 základní jednotky programovacího modelu:

Blok (*thread block*) - Vlákna jsou organizována do 1D, 2D nebo 3D bloků, kde vlákna ve stejném bloku mohou sdílet data a lze synchronizovat jejich běh. Počet vláken na jeden blok je

závislý na výpočetních možnostech zařízení. Každé vlákno je v rámci bloku identifikováno unikátním indexem přístupným ve spuštěném kernelu přes zabudovanou proměnou **threadIdx**.

Mřížka (*grid*) - Bloky jsou organizovány do 1D, 2D nebo 3D mřížky. Blok lze v rámci mřížky identifikovat unikátním indexem přístupným ve spuštěném kernelu přes zabudovanou proměnou blockIdx. Každý blok vláken musí být schopen pracovat nezávisle na ostatních, aby byla umožněna škálovatelnost systému (na GPU s více jádry půjde spustit více bloků paralelně oproti GPU s méně jádry kde bloky poběží v sérii).

Warp - Balík vláken zpracovávaných v jednom okamžiku se nazývá warp. Jeho velikost je závislá na počtu výpočetních jednotek. Počet a organizace spuštěných vláken v jednom bloku a počet a organizace bloků v mřížce se určuje při volání kernelu.

3.2.1.2 Paměťový model

Na grafické kartě je 6 druhů paměti, se kterými může programátor pracovat.

Pole registrů - je umístěno na jednotlivých stream multiprocesorech a jejich rozdělení mezi jednotlivé stream procesory plánuje překladač. Každé vlákno může přistupovat pouze ke svým registrům.

Lokální paměť - je užívána v případě, že dojde k vyčerpání registrů. Tato paměť je také přístupná pouze jednomu vláknu, je ale fyzicky umístěna v globální paměti akcelerátoru a tak je přístup k ní paradoxně pomalejší než např. ke sdílené paměti.

Sdílená paměť - je jedinou pamětí kromě registrů, která je umístěna přímo na čipu streaming multiprocesoru. Mohou k ní přistupovat všechna vlákna v daném bloku. Ke sdílené paměti se přistupuje přes brány zvané banky. Každý bank může zpřístupnit pouze jednu adresu v jednom taktu a v případě, že více streaming procesorů požaduje přístup přes stejný bank, dochází k paměťovým konfliktům, které se řeší prostým čekáním. Speciálním případem je situace, kdy všechna vlákna čtou ze stejné adresy a bank hodnotu zpřístupní v jednom taktu všem streaming procesorům pomocí broadcastu.

Globální paměť - je sdílená mezi všemi streaming multiprocesory a není ukládána do cache paměti.

Paměť konstant - je paměť pouze pro čtení, stejně jako globální paměť je sdílená s tím rozdílem, že je pro ni na čipu multiprocesoru vyhrazena L1 cache. Podobně jako sdílená paměť umožňuje rozesílání výsledku broadcastem.

Paměť textur - je také sdílená mezi streaming multiprocesory, určena pro čtení a disponuje cache pamětí. Je optimalizována pro 2D prostorovou lokalitu, takže vlákna ve stejném warpu, které čtou z blízkých texturovacích souřadnic dosahují nejlepšího výkonu.

3.2.2 Jazyk CUDA C

CUDA C je programovací jazyk se syntaxí C [12]. Koncepčně je docela odlišný od běžného jazyka C. Poskytuje jednoduchou cestu pro uživatele obeznámené s programovacím jazykem C pro snadné psaní programů pro zařízení s podporou CUDA technologie. Jazyk CUDA C obsahuje velmi mnoho nástaveb a rozšíření, které je možné využít pro snadnější implementaci, správu paměti a jiné. V následující ukázce je možné si všimnout značných odlišností od běžného jazyka C a také porovnat s výše uvedenou ukázkou OpenCL, zejména pak zcela jiné volání kernelu. Jinak struktura kódu je s OpenCL totožná. Opět zde máme hlavní funkci Main, ve které se nachází alokace paměti, volání kernelu a přesun dat z hosta do zařízení a vrácení výsledků. Na konci je navíc přidána funkce na vyčištění dříve alokovaných pamětí (viz Ukázka 2).

```
#include "stdafx.h"
#include <stdio.h>
#include <cuda.h>

//Kernel vykonaný v GPU
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}

//část vykonaná na hostu
int main(void)
{
  float *a_h, *a_d;  //ukazatele na hosta a GPU
  const int N = 10;  //počet prvků v poli
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        //přidělení paměti hostovi
  cudaMalloc((void **) &a_d, size);   //přidělení paměti GPU
  //inicializace pole hosta a zkopírování na GPU
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  //provedení výpočtu na GPU
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  //vrácení výsledku z GPU hostovi a uložení v poli
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  //zobrazení výsledků
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  //vyčištění paměti
  free(a_h); cudaFree(a_d);
}
```

Ukázka 2: Práce s polem v CUDA

3.2.3 Požadavky pro práci s NVIDIA CUDA

Základním požadavkem pro práci s technologií CUDA je mít nainstalován nejnovější balíček CUDA SDK 5.5, který podporuje nejnovější grafické karty NVIDIA. Jsou v něm také obsaženy vývojové nástroje Nsight Visual Studio edition pro Windows a Nsight Eclipse Edition pro Linux, MAC OS X a potřebné ovladače a ukázky kódu. Mezi podporované operační systémy patří Windows XP/Vista/7/8, Ubuntu 9.10 a vyšší a Red Hat 5.3 a vyšší.

NVIDIA v současné době poskytuje tři typy GPU produktů s podporou CUDA. Jedná se o grafické karty nVidia GeForce, NVIDIA GeForce Mobile řady 8, 9, 100, 200, 300, 400, 500, 600 a 700, Quadro FX nebo NVIDIA Tesla s nejméně 256 MB grafické paměti.

GeForce - řady GeForce NVIDIA GPU jsou prodávány především prostřednictvím nejrůznějších grafických karet a základních desek výrobců, které obsahují NVIDIA čipy. Výkon bývá velmi vysoky a ceny nízké díky velké poptávce a také díky konkurenci v herních trzích.
Quadro - řada Quadro NVIDIA GPU vyrábí a prodává přímo NVIDIA na velmi high-end profesionální trhy grafiky pracovních stanic. Quadro karty poskytují mimořádně vysoké rozlišení a masivní grafické paměti karty pro nejnáročnější aplikace pracovních stanic. Některé Quadro produkty také se objevují v přenosných počítačích, a jiné jsou zase k dispozici v externích skříních podobně jako u karet Tesla.
Tesla - řada Tesla NVIDIA GPU také vyrábí a prodává přímo NVIDIA na podporu vysoce výkonných počítačů, kde se vyžaduje výkon superpočítače díky paralelnímu zpracování. I když jsou k dispozici také jako plug-in karty , jsou Tesla GPU nejlépe známé pro svou hotovou instalaci do počítačových skříní (buď stolní, nebo do racku), které poskytují dvě nebo čtyři GPU na každou skříň počítače.

Dále je možné využít mnoho podpůrných nástrojů pro vývoj, jako jsou profilovací nástroje (NVIDIA Parallel Nsight, NVIDIA Visual Profiler, TAU Performance systém, VampirTrace, The PAPI CUDA Component), které slouží k optimalizaci kódu nebo nástroje pro debugging (NVIDIA Parallel Nsight, CUDA-GDB, CUDA-MEMCHECK, TotalView, Allinea DDT) pro ladění kódu.

3.2.4 Výhody a nevýhody platformy NVIDIA CUDA

Největší výhodou této technologie je určitě to, že výpočty je v dnešní době možné provádět na většině grafických karet NVIDIA. Při pominutí výkonu jednotlivých karet si každý může volně stáhnout SDK a sám volně vyvíjet. Další výhodou je cenová dostupnost a široká škála těchto karet, pokud pomineme ty nejvýkonnější karty z řady Tesla určené pro

nejnáročnější podmínky použití, které nejsou určené pro běžného uživatele. Dále má CUDA několik výhod oproti tradičním univerzálním výpočtům na GPU (GPGPU) pomocí grafického rozhraní API. Je to například čtení kódu z libovolné části paměti, využité sdílené paměti a její použití pro uživatelské řízení cache umožňuje větší šířku pásma, než je možné s použitím textur vyhledávání, rychlejší kopírování dat z hosta do GPU a zpět a plná podpora pro celočíselné a bitové operace, včetně celo číselných textury vyhledávání. Tedy vlastní definované výrobcem. Jedná se například o matematické knihovny a různé rozšiřující mechanismy.

Obecné výhody pak spočívají ve využití paralelního hardwaru navrženého tak, aby generický (non-grafický kód), s příslušnými ovladači byl vykonán co nejefektivněji. Programovací jazyky založené na C pro programování sdělí hardwaru a sadě instrukcí, že jiné programovací jazyky lze použít jako cíl. Kompletní SDK balíček obsahující knihovny, různé profilovací a debuggovací nástroje, kompilery, které umožní hostovi pomocí programovacího jazyka zavolat GPU.

Mezi nevýhody této technologie patří například to, že CUDA nepodporuje plnou normu jazyka C, tak jako běžný hostitelský kód pomocí kompilátoru C++, který vytváří validní kód C (ale nevalidní C++), který selže při kompilaci. Kopírování mezi hostitelem a pamětí GPU může být předmětem ztráty výkonu vzhledem k šířce pásma systémové sběrnice a latence. Vlákna by měl být spuštěna ve skupinách alespoň po 32 pro nejlepší výkon s celkovým počtem v řádech tisíců. Závorky v kódu programu nemají vliv na výkon výrazně za předpokladu, že každé z 32 vláken má stejné cesty pro provádění a realizaci SIMD modelu, jinak dochází k výrazným omezením (např. při křížení a dělení prostoru datové struktury v průběhu ray tracingu). Další nevýhodou je, že na rozdíl od OpenCL, které je multiplatformní je CUDA omezena pouze na grafické karty NVIDIA. Ve srovnání s OpenCL můžeme některé vlastnosti považovat za proprietární. Funkce prováděné v rámci CUDA karet jsou prakticky okamžité ve srovnání rychlosti realizace výpočtu v rámci host procesoru. Nicméně NVIDIA stream procesory při vykonávání jsou tak rychlé, že je obtížné poskytnout data dostatečně rychle z disku a operační paměti. Výsledné výkonnosti ve většině aplikací jsou proto zpravidla omezena nikoli rychlostí procesoru, ale spíše rychlostí, s jakou údaje lze získat z pevného disku nebo jiných paměti. Kromě toho je velká část různých úkolů, které nejsou vázány výpočtem, ale namísto toho zahrnují režijní úkoly, jako je zápis výsledků na disk a další potřebné úkoly, které procesory CUDA neurychlí.

Nakonec nesmíme také zapomenout na celkovou konfiguraci počítače, ve kterém bude karta instalována, tedy použít adekvátní procesor nejlépe 4 nebo 6 jádrový (respektive mající 8

nebo 12 virtuálních jader), operační paměť minimálně 4 GB, napájecí zdroj, aby bylo možné dosáhnout maximálního potenciálu grafické karty.

3.2.5 Budoucnost CUDA

Grafické karty dnešní doby zaznamenávají každým rokem razantní nárůst výkonu, kterého si vědecká komunita nemohla nevšimnout a začala je využívat pro všeobecné použití výpočetní techniky. Ukazuje se, že mnohé matematické výpočty, jako je násobení matice a transpozice, které jsou nezbytné pro komplexní vizuální a fyzikální simulace ve hrách jsou totožné s výpočty, které musí být provedeny v nejrůznějších vědeckých výpočetních aplikacích, včetně GIS.

NVIDIA se v dalších letech plánuje zaměřit na vývoj hlavních 4 částí této platformy. První z nich je vylepšení stávajících kompilerů (C++ new/delete virtuální funkce). Druhou částí směru vývoje je rozmanitost použití programovacích jazyků (UVA, OpenACC) a vývoj novější verze C++. Třetí částí je pak rozšíření stávajících knihoven a vývoj nových knihoven (cuBLAS, Device API). Posledním ze 4 cílů je zlepšení profilovacích a debuggovacích nástrojů. Hlavním cílem je pak použití GPU napříč všemi technologiemi, nahradit stávající v porovnání nevýkonné CPU.

4 Intel Xeon Phi softwarová architektura

Tato kapitola pojednává o softwarové architektuře platformy Intel Xeon Phi. Popisuje, za jakých podmínek lze dosáhnout nejvyššího výpočetního výkonu, srovnání s GPU, ale také již názorné ukázky kompilace a spouštění jednoduchých aplikací. V roce 2013 byly vydány první knihy [1], [2], [3] zabývající se touto problematikou programování pro Intel Xeon Phi. Rovněž existuje spousta vývojářských fór [4], [5] a uživatelské příručky Intelu [6].

4.1 Základní principy

Většina aplikací na světě nebyla strukturována tak, aby využívala paralelismů. To ponechává velké množství nevyužitých možností téměř na každém počítačovém systému. Tyto aplikace mohou být rozšířeny po výkonnostní stránce vysoce paralelním zařízením pouze tehdy, když aplikace vyjadřují potřebu paralelismu pomocí paralelního programování. Pokyny pro úspěšné paralelní programování lze shrnout jako „Program se spoustou vláken, který používá vektory s Vámi preferovaným programovacím jazykem a paralelními modely.“ Dvojitou výhodou Intel Xeonu Phi je použití stejných paralelních programovacích modelů, programovacích jazyků a známých nástrojů, které výrazně zvýší zachování programovacích investic.

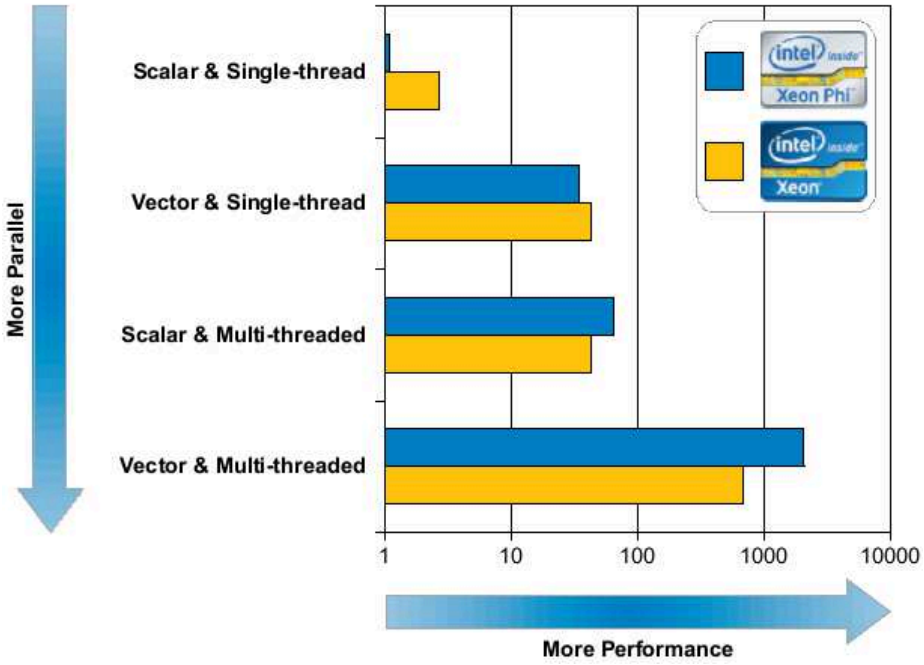
Kompletní architekturu koprocesoru Intel Xeon Phi byla popsána dříve. Nyní se zkusme zaměřit na jeho softwarovou stránku z pohledu programátora. Jaké má vlastnosti, výhody či nevýhody a alespoň částečné porovnání s jinými platformami.

Koprocesory postavené na čipech Knights Corner mohou poskytnout více než 1 GFLOPs v dvojité pohyblivé řádové čárce a více než 2 GFLOPs v jednoduché pohyblivé řádové čárce. Programátoři mohou dosáhnout této úrovně superpočítačů ohromné výpočetní síly několika způsoby:

1. Použitím pragmat k rozšíření stávajícího kódu, k přesunutí vykonávané úlohy z hostitelského procesoru na koprocesor (offload).
2. Překompilování zdrojového kódu pro spuštění přímo v koprocesoru, jako samostatný mnoho-jádrový Linux SMP výpočetní uzel (native).
3. Přístup ke koprocesoru jako akcelerátoru pomocí optimalizovaných knihoven, jako je Intel MKL (Math Kernel Library).
4. Použití každého z koprocesorů jako uzel v clusteru MPI nebo alternativně jako zařízení obsahující shluk MPI uzlů.

4.2 Dosažení nejvyššího výkonu koprocesoru

Z tohoto seznamu vyplývá, že Intel Xeon Phi koprocesor podporuje celou škálu moderních i starších programovacích modelů. Zjišťujeme tak, že můžeme koprocesor naprogramovat v podstatě stejným způsobem jako stávající x86 systémy. Klíč spočívá ve vyjádření dostatečného paralelismu a vektorové schopnosti dosáhnout vysokého výkonu s plovoucí desetinnou čárkou, protože Intel Xeon Phi koprocesor poskytuje více než řádové zvýšení počtu jader oproti současným například 4 jádrovým procesorům. Masivní vektorový paralelismus, je skutečně cesta k dosažení vysokého výpočetního výkonu viz Obrázek 15.



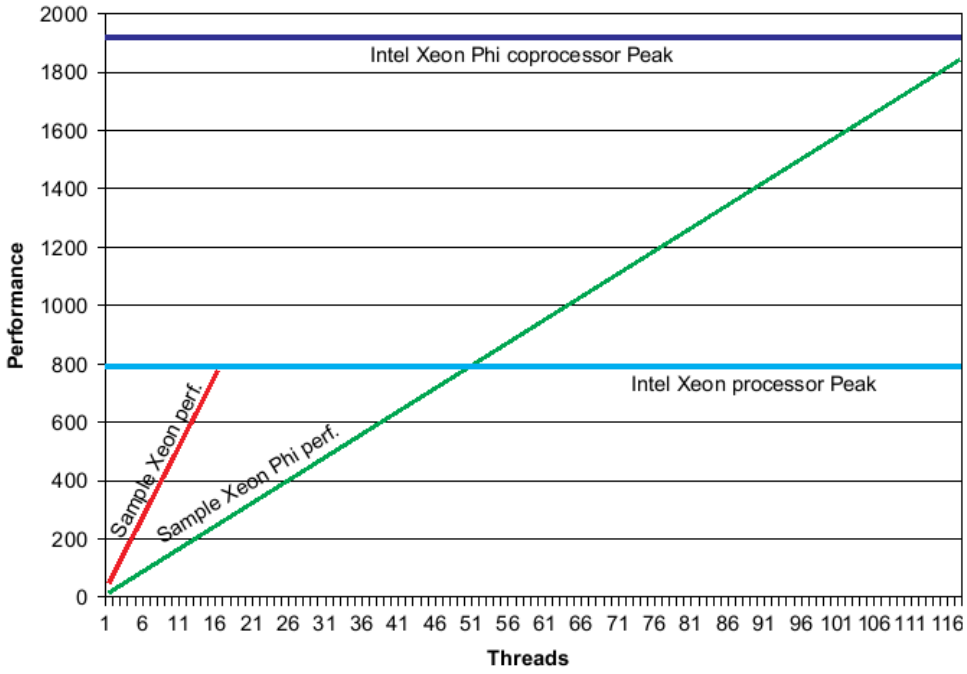
Obrázek 15: Využití paralelismu i vektorového zpracování u Xeonu Phi v aplikacích [1]

Pro dosažené vysokého výpočetního výkonu s externím koprocesorem je nutné, aby programátor zajistil uskutečnění hlavních 3 kroků:

- Přenesení dat po PCIe sběrnici na koprocesor a jejich uchování v paměti
- Dát koprocesoru práci – s jakými daty má pracovat a co s nimi má dělat
- Zamezení opakovanému použití dat v koprocesoru, aby se zabránilo překážkám v propustnosti paměti a přesunu dat zpět do hostitelského procesoru

Zatímco celkový Intel Xeon Phi výpočetní výkon je vysoký, každé jádro je pomalé a má omezený výkon s plovoucí desetinnou čárkou v porovnání s moderními procesory Intel Core postavené na architektuře Haswell. Vysokého výkonu lze dosáhnout pouze tehdy, když se použije velký počet paralelních vláken (minimálně 120) a pokyny ke zpracování instrukcí jsou

posílány do VPU dostatečně rychle, aby byly vektorové pipelines stále plné. Současná generace jader koprocesoru podporuje vykonávání až čtyř souběžných vláken přes hyperthreading. Většina vývojářů se bude s jistotou spoléhat na kompiler, že rozpozná, kdy mohou být použity speciální vektorové instrukce základních vektorových jednotek. Alternativou je použít kompilátor vnitřních operací nebo jazyk symbolických instrukcí pro přístup k vektorové jednotce. To znamená, že stávající knihovny a aplikace musí být znovu překompilovány, aby správně běžely na Intel Xeonu Phi. Obecně platí, že nejlepší výkon s plovoucí desetinnou čárkou se realizuje, když na každém jádře běží dvě vlákna, která aktivně posílají instrukce vektorové jednotce. Pro 61-jádrový koprocesor to znamená, že programátor musí být schopen efektivně využít 120 vláken (dvakrát počet jader mínus jedno hlavní vyhrazeno pro operační systém) v dané aplikaci. Obrázek 16 ukazuje rozdíl mezi použitím paralelní aplikace na běžném procesoru Intel Xeonu a koprocesoru Intel Xeon Phi, je vidět, že s rostoucím počtem vykonávaných vláken rapidně roste i výkon.



Obrázek 16: Teoretický výkon Intel Xeonu Phi [1]

Dalším způsobem dosažení ještě vyššího výkonu je využití transformací. Existuje celá řada možností optimalizace na straně programátora, které jsou účinné pro dosažení maximálního výkonu. Tyto pokročilé techniky nejsou povinné, ale jedná se o další způsob, jak z koprocesoru extrahovat ještě další výkon pro aplikaci. Je nepravděpodobné, že špičkového výkonu bude dosaženo, aniž by se přihlédlo k některým s těchto optimalizací:

- Přístup k paměti a transformace cyklů (např. blokování cache, rozbalování cyklů, prefetching, výměna cyklů, zarovnávání).
- Vektorizace funguje nejlépe na dílčích krocích vektorů. Transformace datových struktur může zvýšit množství dat s přístupem přes dílčí kroky (AoS3, SoA4 transformace nebo překódování k použití zabalených namísto nepřímých přístupů).
- Použití úplných (nikoli jen částečných) vektorů je nejlepší. Transformaci dat je třeba pořádně promyslet k dosažení nejlepších cílů.
- Vektorizace je nejlepší provádět se správně uspořádanými daty.
- Ve výběru algoritmů upřednostňovat ty, ve kterých je možné uplatnit paralelizaci a vektorizaci.

Vektor je skupina datových položek stejného datového typu, které mohou být zpracovány paralelně jedinou instrukcí. Vektor je obvykle generován kompilerem a to tak, že změní pole dat do formátu vektoru, který je podporován základním zpracováním architektury.

Intel Xeon Phi využívá multi-threadingu na každém jádře, jako způsob zamaskování latencí způsobených mikroarchitekturou. To by nemělo být zaměňováno s hyper-threadingem procosorů Intel Xeon, který je primárně určen pro lepší přenos dat a dynamické vykonávání operací. Zatímco na běžných Intel Xeon procesorech je možné hyper-threading ignorovat nebo i vypnout, na Intel Xeonu Phi to možné není. Mutli-threading programů by neměl být ignorován a hardwarové vlákna nelze vypnout.

Aplikace mohou používat jak procesor Intel Xeon tak Intel Xeon Phi koprocesor současně a přispívat k výkonu aplikace. Aplikace by měla využít koprocesor pro zpracování, pokud to může přispět k výkonnosti celého výpočetního uzlu. Obecně lze říci, že bude využit v průběhu části aplikace, která bude využívat vysoký stupeň paralelismu. U některých zatížení může koprocesor přispívat podstatně větším výkonem než samotný procesor, zatímco v jiných případech tomu může být naopak. Vzhledem k sofistikovanému řízení spotřeby u procesoru Intel Xeon a Intel Xeon Phi koprocesoru, lze výkonovou účinnost uzlu zachovat v širokém spektru aplikací tím, že spotřebovávají energii pouze v případě potřeby přispět k výkonnosti uzlu. Prvním krokem ke správnému využití výkonu Xeonu Phi je maximálně využít potenciál, který aplikace může získat z hostitelského procesoru Intel Xeon. Snažit se využívat Xeon Phi aniž by se maximalizovalo využití paralelismu na Intel Xeonu bude téměř jistě zklamáním z výsledného výkonu.

Klíčem k výpočetnímu výkonu Xeonu Phi v plovoucí desetinné čárce je efektivní využití VPU v každém jádře. Pro přístup k VPU musí být kompilátor schopen rozpoznat SSE

kompatibilní konstrukce, aby bylo možné vytvářet speciální Xeon Phi vektorové instrukce. Programátor si pak může otestovat, zda aplikace skutečně využívá Xeonu Phi a jeho výkonu v plovoucí desetinné čárce, tedy že bylo dosaženo nějakého výraznějšího zlepšení oproti výpočtu na hostitelském x86 procesoru s použitím SSE instrukcí (GNU `-msse` nebo jiným přepínačem kompileru). Aplikace spuštěná rychleji s SSE (nebo naopak zpomalená v případě zakázání používání instrukcí SSE) bude těžit z Xeonu Phi VPU. Aplikace, která nemá prospěch z instrukční sady SSE bude omezena na výkonnosti jednotlivých jader. Z toho jasně vyplývá, že Xeon Phi není nejvýkonnější pro non-vektorové aplikace, avšak pro vektorové aplikace s využitím mnoho-jádrového paralelismu a vysoké propustnosti pamětí je vhodným řešením.

Při výpočtech v jednoduché pohyblivé řádové čárce je možné výpočet provádět přes 16 řad (32 bitů) a při dvojité pohyblivé řádové čárce přes 8 řad (64 bitů). Metoda stanovení výkonové špičky v jednoduché a dvojité pohyblivé řádové čárce Intel Xeonu Phi je:

$$Frekvence \times Počet\ jader \times počet\ řad \times \frac{2(FMA)FLOPs}{cyklus}$$

Pro mnou testovaný koprocesor 5110P je to tedy:

$$1{,}053\ GHz \times 60 \times 16 \times 2 = 2021{,}8\ GFLOPs$$

$$1{,}053 \times 60 \times 8 \times 2 = 1010{,}9\ GFLOPs$$

4.3 Programovací jazyky

Žádný z programovacích jazyků nebyl navržen přímo pro paralelismus. Intel Xeon Phi nabízí plnou podporu použití stejných programovacích nástrojů, jazyků a modelů jako procesory Intel Xeon. Nicméně vzhledem k vysokému stupni paralelismu jsou některé modely mnohem důležitější než u běžných procesorů. Pro Fortran programátory je vhodné použití OpenMP a MPI. Pro C11 programátory je vhodné použití Intel TBB, Intel Cilk Plus a OpenMP. Pro programátory C je vhodné použít OpenMP a Intel Cilk Plus. Intel TBB je C11 template knihovna, která nabízí vynikající podporu pro vyvažování zátěže. Intel TBB je open source a je k dispozici na celé řadě platforem podporovaných většinou operačních systémů a procesorů. Intel Cilk Plus je trochu složitější v tom, že nabízí jak tasking a vektorizaci. Programátoři využívající sdílenou paměť mají velmi doporučené použití Intel TBB a Intel Cilk Plus. Intel TBB má rozšířené možnosti použití v C11 komunitě programátorů a Intel Cilk Plus zase rozšiřuje Intel TBB řešení a vektorizaci v C a C11 programech. Naštěstí Intel Cilk Plus plně spolupracuje s Intel TBB. Intel Cilk Plus nabízí jednodušší sadu task funkcí než Intel TBB, ale

pomocí klíčových slov v jazyce je možné získat plnou podporu kompileru. V současné době je Intel Cilk Plus k dispozici od společnosti Intel pro Windows, Linux a Apple OS X.

4.4 Optimalizace cache

Nejefektivnější způsob využití cache je věnovat pozornost umístění odkazů, blokování, a aby se vešly do L2 cache. Důležité je také zajistit, aby byl využíván prefetching (hardwarem, překladačem, knihovnou). Uspořádání data tak, aby se vešly do 512 KB nebo menší L2 cache jádra dává nejvyšší možný způsob využití cache. Všechny čtyři hardwarové vlákna každého jádra se podělí se o svou lokální "per core" L2 cache, protože mají vysokorychlostní přístup do cache a jsou spojeny s ostatními jádry. Veškeré údaje použité konkrétním jádrem budou zabírat prostor v této lokální L2 cache. Všimněme si přínosu sdílení vláken jádra, kde bychom neměli očekávat výkonnostní zlepšení podle toho, jak blízko je jedno jádro druhému na koprocesoru. I když se to může zdát překvapivé, hardwarový design je tak dobrý, že v tomto ohledu nezjistíme žádné znatelné zvýšení výkonu na základě blízkosti jader v koprocesoru. Proto se nedoporučuje zbytečně trávit čas nad snahou toto zohledňovat v programu. Koprocesor má hardwarový prefetching do L2, který je zahájen první chybějící cache v rámci stránky. Intel kompilátory řeší problém software prefetche agresivně pro paměťové odkazy uvnitř cyklů ve výchozím nastavení. Prefetch vzdálenost vypočítá kompilátor na základě množství práce uvnitř cyklu. Explicitní prefetch může být přidán buď pro načítání pragmat (#pragma prefetch v C/C11 nebo CDec $ prefetch ve Fortranu). Explicitně je také možné vypnout prefetching přímo v kompilátoru (-opt-prefetch = 0 pro vypnutí prefetching, nebo -opt-prefetch-distance = 0,2 vypne kompilátoru prefetching do L2). Právě kvůli nastavení cache mohou být některé paměťové transformace použity opětovně například pro další ladění, organizaci datových proudů nebo organizaci dat.

4.5 Porovnání koprocesoru s GPU

Stručně řečeno GPU nemůže nabídnout programovatelnost koprocesoru Intel Xeonu Phi, protože nesdílejí podmnožinu věcí, které by bylo možné urychlit pomocí škálování v kombinaci s vektorizací a propustností. Jinými slovy aplikace, která dokáže těžit z výkonu GPU, dokáže vždy těžit i z koprocesoru, protože musí být zachovány stejné podmínky vektorizace a propustnosti. Opačně to však nefunguje. Flexibilita koprocesoru totiž zahrnuje i aplikace, které na GPU spustit nelze. To je jeden z hlavních důvodů, proč systém obsahující Intel Xeon Phi koprocesor bude mít širší využití než systém s GPU. Navíc ladění aplikace pro GPU je obvykle velmi odlišné od procesoru, takže prospěch můžeme vidět v programování pro koprocesory

Intel Xeon Phi. To může vést k podstatnému snížení investic díky přenositelnosti na jiné systémy.

Jak již bylo řečeno výše v kapitolách o platformách OpenCL a CUDA, OpenCL je tak univerzální, že je možné jej spouštět na grafické kartě jako CUDA aplikaci. Něco podobného platí i v případě Phi koprocesoru, na kterém můžeme spustit CUDA aplikace, tedy i OpenCL aplikace. CUDA i Phi koprocesor poskytují vysoký stupeň paralelismu, která může zajistit vynikající výkon aplikací. Nicméně pro dosažení nejvyššího výkonu je třeba učinit několik úprav. CUDA aplikace může běžet na hardwaru x86, ale je důležité vědět, jaké jsou rozdíly v architektuře mezi GPU a Intel Xeon Phi koprocesory a jak to ovlivní výkon a design aplikace.

Offload režim koprocesoru je odrazem toho, jak CUDA programátoři v současné době využívají GPU zařízení. V offload režimu musí být data přesunuta po PCIe sběrnici do externího zařízení stejně jako při programování GPU. Podobnost je také v optimalizované MKL knihovně koprocesoru, která nabízí práci s nativním rozhraním (kde jsou data uložena nativně na zařízení) a tzv. thunking rozhraní (kde jsou data přenášena po sběrnici PCIe pro každou operaci) stejně jako CUBLAS a CUFFT.

Obě zařízení Intel Xeon Phi a GPU může akcelerovat MPI (Message Passing Interface) aplikace v offload režimu, ve kterém jsou části aplikace akcelerovány externím zařízením. Nicméně CUDA programátoři si musí uvědomit, že mohou také spustit MPI a OpenMP kód nativně na koprocesoru. Při běhu MPI nativně může každý koprocesor působit jako samostatný SMP uzel v distribuované aplikaci MPI.

Z hlediska vývojových nástrojů je na tom lépe samozřejmě platforma CUDA, protože je zde již několik let (viz kapitola CUDA výše). Vývojových nástrojů pro Intel Xeon Phi v současnosti není mnoho, ale vývoj dalších probíhá. Většina jich je právě od Intelu, například kompilery Intel Parallel Studio XE, Intel C++ Composer XE, Intel Fortran Compiler XE. Jako vhodný nástroj pro profiling je možné použít Intel VTune Amplifier a jako debuggovací nástroje Intel Inspector nebo GNU Project Debugger. Dále jsou zde nástroje pro mapování CUDA aplikací na Intel Xeon Phi.

V současné době je pro spouštění CUDA aplikací na Intel Xeon Phi vyžadováno provést pár změn ručně. I když je technicky možné vykonat CUDA kód na koprocesoru až po těchto změnách, ani CUDA-86 není přímo určena pro vykonání na koprocesoru. Aktuálně je ve vývoji OpenCL kompiler (source translator), který přímo převede OpenCL kód do podoby, aby jej bylo možné snadno spustit na koprocesoru. Tím se otevírá okno i pro CUDA programátory, kteří mohou použít například Wu Feng's CU2CL CUDA to OpenCL source translator. I když se to

zdá jako poměrně složitá oklika, je to zatím jediný způsob automatického překladu kódu z OpenCL a CUDA pro Intel Xeon Phi. V budoucnu určitě přibyde i LLVM přímý překladač CUDA kódu pro Intel Xeon Phi.

Tabulka 1 níže uvádí stručný přehled o podobnosti a rozdílech mezi platformou CUDA a Intel Xeon Phi v programovacích modelech a možných přístupech (kromě ručního překladu).

Programovací přístup	**Intel Xeon Phi koprocesor**	**CUDA-enabled zařízení**
Jazyky jako C / C++ / Fortran atd.	Nativní i offload režim, ale vyžaduje použití vláknového modelu jako Pthreads nebo OpenMP	Jedině prostřednictvím offload režimu programování. Mnoho jazyků lze urychlit pouze voláním CUDA, OpenCL nebo metod knihoven
CUDA, OpenCL akcelerované	Offload režim. Blíží se podpora OpenCL překladače. Alternativními cestami jsou: CU2CL CUDA to OpenCL source translator LLVM překlad Manuální překlad	Přímo na zařízení jako offload akcelerované
Directive-based programování	Přes OpenMP nativní i offload režim	Pouze přes OpenACC jako externí akcelerátor
Programování s knihovnami	Nativní i offload režim	Nativní i offload režim
MPI	Nativní i offload režim	Pouze offload režim

Tabulka 1: Podobnosti a rozdíly mezi Intel Xeon Phi a CUDA zařízeními

4.5.1 Rozdíl mezi vláknem Xeonu Phi a CUDA

CUDA a Intel Xeon Phi aplikace využívají velký počet souběžných vláken, které využívají hardwarového paralelismu k dosažení vysoké výkonnosti. Je však důležité pochopit rozdíly mezi CUDA a Intel Xeon Phi vlákny.

Z programátorského hlediska, jak CUDA tak Intel Xeon Phi vlákna jsou oblastí sériového kódu, které byly identifikované programátorem nebo kompilátorem jako kandidáti pro paralelní vykonání. Je už ale na zařízení aby rozhodlo, jak budou paralelní vlákna probíhat.

Každá sériová část kódu obsažená ve vlákně je výpočetně univerzální, což znamená, že každá výpočetní funkce může být teoreticky provedena v rámci jednotlivých CUDA nebo Intel Xeon Phi vláken. Jinými slovy, nejsou kladena žádná omezení na typ výpočtu, který lze vyjádřit v rámci jednoho vlákna. Programátor může ve vlákně použít jakýkoliv kód chce. To ale neznamená, že vlákna CUDA a Phi koprocesoru poskytují rovnocenné parametry a výkon v paralelním prostředí.

Individuální programovatelnost vláken CUDA a Phi koprocesoru lze vidět na následujících dvou příkladech, které ukazují jedno vlákno „Hello World“. Tyto dva příklady jsou uvedeny k prokázání obecné programovatelnost jednotlivých vláken CUDA a Phi koprocesoru. Skutečná hodnota programování na GPU a Phi koprocesor spočívá ve výkonu těchto zařízení, kterého můžeme dosáhnout při spuštění velkého počtu paralelních vláken.

První příklad je kód CUDA aplikace spouštěný výhradně v offload režimu a pod ním samotný výstup po zkompilování a spuštění (viz Ukázky 3 a 4). Všimněme si způsobu deklarace funkcí a volání kernelu.

```
#include <stdio.h>

__device__ void toBinary(char* buf, int d)
{
  for(int i=0; i < 8*sizeof(int); i++)
     buf[i] = ((d>>i)&0x1)?'1':'0';
  buf[8*sizeof(int)] = 0;
}

__global__ void hello(int d)
{
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  char *s="Hello World";
  char buf[8*sizeof(int)+1];
  toBinary(buf,d);

  printf("%s thread %d, d=%d binary %s\n",s, tid, d, buf);

}

int main()
{
  const int nThreads=1, nBlocks=1;
  hello<<<nBlocks, nThreads>>>(5);
  cudaThreadSynchronize();
  return 0;
}
```

Ukázka 3: Kód CUDA aplikace Hello World

```
$ nvcc -arch=sm_20 hello.cu -run
Hello World thread 0, d=5 binary 10100000000000000000000000000000
```

Ukázka 4: Kompilace, spuštění a výstup CUDA aplikace

Ukázka 5 je aplikace „Hello World“ která může běžet nativně nebo v offload režimu na koprocesoru Phi. Při offload režimu jsou to právě pragmata, která určují, jaké metody a cykly musí být provedeny na koprocesoru.

```
#include <omp.h>
#include <stdio.h>

#pragma offload_attribute (push, target (mic))

void toBinary(char* buf, int d)
{
  for(int i=0; i < 8*sizeof(int); i++)
     buf[i] = ((d>>i)&0x1)?'1':'0';
  buf[8*sizeof(int)] = 0;
}

void hello(int d)
{
  int tid = omp_get_thread_num();
  char *s="Hello World";
  char buf[8*sizeof(int)+1];
  toBinary(buf,d);

  printf("%s thread %d, d=%d binary %s\n",s, tid, d, buf);

}
#pragma offload_attribute (pop)

int main()
{
  const int nThreads=1;
  #pragma offload target(mic)
  #pragma omp parallel for
  for(int i=0; i < nThreads; i++) hello(5);
  return 0;
}
```

Ukázka 5: Kód Hello World aplikace pro koprocesor Xeon Phi

Ukázka 6 má dva příkazy, které ukazují, jak zkompilovat Ukázku 5 pro nativní vykonání a offload režim.

```
$ icc -mmic -openmp -Wno-unknown-pragmas -std=c99 hello.c
$ icc -std=c99 hello.c
Hello World thread 0, d=5 binary 10100000000000000000000000000000
```

Ukázka 6: Kompilace, spuštění a výstup koprocesoru Phi

4.5.2 CUDA a Xeon Phi výkonávání paralelních vláken

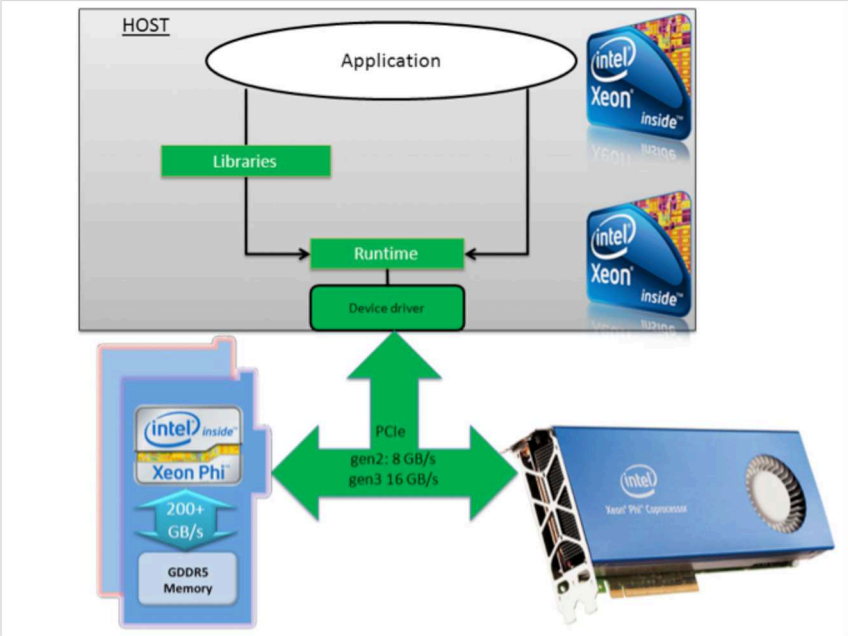
I když programování jediného vlákna CUDA nebo Phi koprocesoru je podobné, může být naprogramování více vláken paralelně velmi odlišné. Tyto rozdíly pramení ze skutečnosti, že CUDA vlákna musí být seskupeny do bloků (tzv. "thread blocks" v CUDA a "work groups" v OpenCL), které se provádějí souběžně v GPU Streaming multiprocesoru podle SIMD modelu, zatímco Phi koprocesory spouští MIMD vlákna individuálně na x86 jádrech. Volnost provádění MIMD umožňuje vykonat jakékoliv vlákno, jakoukoliv instrukci v jakémkoliv čase. To se

oproti GPU jeví jako jasná výhoda. Z tohoto důvodu GPU zařízení rovněž podporují MIMD provedení tím, že každý blok vláken, který má být naplánován v každém SM na GPU k vykonání, umožňuje GPU provést mnoho různých instrukcí paralelně ve stejnou dobu. Takto modifikovaná forma GPU MIMD je označována jako SIMT (jedna instrukce více vláken). Rozdíl mezi vykonáváním MIMD a SIMT vláken může způsobit, že některé aplikace běží rychleji (potenciálně mnohem rychleji) na jedné hardwarové platformě nebo na druhé.

Jsou zde však jistá kritéria, které musí programátor dodržet, aby aplikace běžela paralelně. Xeon Phi koprocesor používá MIMD vlákna, která nekladou žádné omezení na to, jak vlákna spustit nebo jak komunikují. Je však povinností programátora zajistit, že vlákna nejsou blokované, jsou zadány všechny vstupní parametry a není omezená škálovatelnost. OpenMP model založený na pragmatech zjednodušuje použití takovýchto vláken. Cuda programátoři jsou zase povinni seskupovat vlákna do bloků. Všechny vlákna nezařazené do bloků budou provedena jako SIMD model, protože pouze vlákna zařazené do bloků mohou mezi sebou komunikovat navzájem. MIMD model je pak možné spustit na základě členitosti bloků a je nutné jej spustit ručně programátorem. CUDA a OpenCL jsou platformní brány pro programování GPU architektury. OpenACC je nový přístup založený na pragmatech, který zjednodušuje používání GPU vláken.

4.6 Výhody a nevýhody platformy Xeon Phi

Asi největším omezením karet první generace je kapacita pamětí, která je pravděpodobně hlavním omezením pro současnou generaci Xeonů Phi. Toto omezené se projeví například v případě spouštění nativní SMP nebo MPI aplikací přímo na koprocesoru. Dalším významným omezením při práci s koprocesorem je poměrně nízká propustnost PCIe sběrnice hostitelského počítače. Aktuální PCIe sběrnice komplikuje práci se složeným programovacím modelem a jeho externím rozdělováním dat. Předpoklad vychází z SMP exekučního modelu, kde každé vlákno může přistupovat k libovolným datům ve sdílené paměti systému bez významnějšího snížení výkonu. Jak je vidět na Obrázku 17, propustnost PCIe sběrnice je podstatně nižší než u vestavěné (on-board) paměti koprocesoru.



Obrázek 17: Propustnost vestavěné paměti koprocesoru a PCIe sběrnice [1]

V porovnání s platformou CUDA, která je dnes cenově dostupná na většině grafických karet NVIDIA, kdy téměř každý může sám zkoušet vyvíjet nějaké aplikace. U Xeonu Phi tomu tak není. Tento koprocesor není určen pro volný prodej, ale pouze pro uzavřený trh jako jsou výzkumné instituce, školy a podobně. Dalším omezením pro jeho širší použití může být i cena, která se pohybuje u nejslabší verze 31xx přes 50 tisíc Korun a budoucí nejsilnější verze vyjde v přepočtu na 130 tisíc Korun.

Mezi výhody patří nesporně snadná a všestranná programovatelnost, kdy na koprocesoru můžeme spustit výpočetní aplikaci napsanou ve velkém množství jazyků. Možnost vykonávání v nativním nebo offload režimu či spouštění CUDA, OpenCL aplikací a mnoho dalších.

Nesmíme také zapomenout na možnost použití v superpočítačích, kde mají tyto karty velký potenciál. Aktuálně nejvýkonnějším superpočítačem Tianhe-2 (Milky Way 2), který se nachází v čínské Národní univerzitě obranných technologií a skládá se z 32 000 procesorů Intel Xeon a 48 000 koprocesorů Intel Xeon Phi, díky nimž dosahuje maximálního výkonu 54,9 PFLOPS (Rpeak) a 33,9 PFLOPS (Rmax). Což je téměř dvojnásobný výkon oproti svému předchůdci Titan - Cray XK7 v americkém Oak Ridge, který je sestaven z procesorů AMD Opteron a NVIDIA Tesla K20X s výkonem 27,1 PFLOPS (Rpeak) a 17,6 PFLOPS (Rmax). Zajímavé také je, že od roku 1997 to je první superpočítač na prvním místě v TOP 500, jenž je založen výhradně na procesorech a koprocesorech Intel.

4.7 Budoucnost Intel Xeonu Phi

Intel na své alternativě k výpočetním GPU dále pracuje a v současnosti jsou uváděny na trh nové verze karet. Nejde ale ještě o karty s novými 14nm čipy Knights Landing, nýbrž o nové modely stávající generace, postavené na 22nm čipech Knights Corner. Tyto karty mají mnohem více paměti a jsou tak schopny beze ztrát výkonu pracovat s většími objemy dat a zvládnou díky tomu zase o trochu náročnější úlohy. V oblasti, do které Xeon Phi míří, může být přitom nedostatečná velikost paměti větším problémem než u grafických karet. Jedním z důvodů rozšíření pamětí je reakce na konkurenční NVIDIA Tesla K40, která by měla mít 12 GB pamětí, než bude vydána nová řada s čipy Knights Landing. Tyto nové čipy jsou výkonnější s vyšším taktem každého jádra, budou mít více pamětí a díky 14nm výrobnímu procesu budou i úspornější než současné karty s 225 nebo 245W TDP.

Nová generace karet s označením Knights Landing by podle dostupných informací měla přinést zásadní změnu ve využití čipů. Mělo by se jednat již o plnohodnotné výpočetní jednotky do LGA slotu jak ho známe nyní s až trojnásobným výkonem oproti současným čipům Knights Corner. Tím se dostáváme na teoretickou výpočetní kapacitu přes 3 TFLOPS při dvojité přesnosti (v jednoduché by to měl být dvojnásobek, vzhledem k tomu, jak SIMD instrukce fungují). Čipy budou mít přitom nižší spotřebu TDP, které má být 160–200 W podle modelu. Počet jader naroste z 62 na 72 a zároveň dvě jednotky AVX-512 znamenají dvojnásobný maximální výkon. Zatímco 22nm Xeon Phi potřebuje překompilovaný software, neboť nepodporuje žádné instrukční rozšíření architektury x86, Knights Landing poskytne plnou binární kompatibilitu s moderními procesory. Novinkou by měla být také ntegrovaná paměť (označená MCDRAM), která má čipu poskytovat propustnost až 500 GB/s a bude mít kapacitu 8 nebo 16 GB GDDR5. Poslední známou důležitou změnou pak bude integrovaný řadič PCI-E 3.0 o 36 linkách. Xeony Phi by mohly znamenat skutečně mnoho pro superpočítače a další nasazení, kde je třeba zejména hrubý aritmetický výkon. Oproti grafickým kartám jsou totiž snáze programovatelné. Tím je možné dosáhnout vyššího reálného výkonu a přiblížit se tak k hodnotám teoretického výkonu. A v neposlední řadě budou čipy Knights Landing schopny pracovat s nepoměrně větším paměťovým prostorem než výpočetní grafické karty.

5 Práce s koprocesorem

Nyní se dostáváme do části, kdy již můžeme začít pracovat s koprocesorem Xeon Phi. Karta se nachází na školním serveru, kde je nainstalován Intel C/C++ Compiler, jsou připojeny knihovny MKL, OpenMP a MPI a je připravený testovací účet pro tyto účely [8], [19], [20]. Ukážeme si, jak nastavit vzdálené SSH připojení, vytvoříme aplikaci, kterou otestujeme ve všech 3 režimech (nativní, offload, hostitelský) a změříme výsledné výpočetní časy a výpočetní výkon této karty v porovnání s procesorem Intel Xeon.

5.1 Konfigurace připojení

Před prvním připojením ke vzdálenému serveru, na kterém se koprocesor nachází, je nutné provést několik nastavení, aby bylo připojení ověřené a bezpečné [13]. K této prvotní konfiguraci je třeba administrátorských oprávnění, protože je nutné uložit novou konfiguraci a provést restart koprocesoru. Po připojení na VPN jsme vyzváni k zadání autorizačních údajů. Následně je nám vytvořen pracovní adresář `/home/username`. Nyní je nutné vygenerovat dvojici privátní/veřejný RSA klíče pro konkrétního uživatele a veřejný nahrát do koprocesoru, protože přístup mají pouze ověření SSH uživatelé. To provedeme příkazem `ssh-keygen`, který uloží klíče do složky .ssh v pracovním adresáři uživatele. Poté veřejný klíč přesuneme do složky `/opt/intel/mic/filesystem/mic0/home/fei/bar0088/.ssh` a tím aktualizujeme jeho image. Posledním krokem je provedení restartu koprocesoru, aby se zkopíroval nově upravený image na koprocesor. Provedeme to následujícími příkazy jako administrátor (viz Ukázka 7):

```
bar0088@argexpr2:~> sudo service mpss stop
bar0088@argexpr2:~> sudo micctrl –resetconfig
bar0088@argexpr2:~> sudo service mpss start
```

Ukázka 7: Restartování koprocesoru

Po provedeném restartu můžeme ověřit připojení ke koprocesoru některým ze dvou příkazů (viz Ukázka 8), kde se přes SSH připojíme na název zařízení (mic0 až micX) nebo přímo pod uživatelským jménem a IP adresou zařízení (172.31.1.1 až 172.31.X.1). V ukázce 9 vidíme kompletní výpis konfigurace připojení, které bylo právě popsáno.

```
bar0088@argexpr2:~> ssh mic0
bar0088@argexpr2:~> ssh bar0088@172.31.1.1
```

Ukázka 8: Připojení ke koprocesoru přes SSH

```
Using username "bar0088".
Using keyboard-interactive authentication.
Password:
Creating directory '/home/fei/bar0088'.
Last login: Tue Feb 25 14:34:11 2014 from w117-121.vsb.cz
bar0088@argexpr2:~> ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/fei/bar0088/.ssh/id_rsa):
Created directory '/home/fei/bar0088/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/fei/bar0088/.ssh/id_rsa.
Your public key has been saved in /home/fei/bar0088/.ssh/id_rsa.pub.
bar0088@argexpr2:~> cd .ssh
bar0088@argexpr2:~/.ssh> cat id_rsa.pub >> authorized_keys
bar0088@argexpr2:~/.ssh> cd
bar0088@argexpr2:~> ssh mic0
The authenticity of host 'mic0 (172.31.1.1)' can't be established.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'mic0,172.31.1.1' (RSA) to the list of
known hosts.
[bar0088@cct407-6-mic0 bar0088]$ hostname
cct407-6-mic0
[bar0088@cct407-6-mic0 bar0088]$ cat /etc/issue
Intel MIC Platform Software Stack release 2.1
Kernel 2.6.38.8-g2593b11 on an k1om
```

Ukázka 9: Celá konfigurace připojení ke koprocesoru

Nyní máme připravené připojení pro další práci a můžeme začít testovat. Můžeme se ještě podívat na detailní informace o koprocesoru, hostovi a pamětech, které lze získat z virtuálního Linuxu pomocí následujících příkazů:

```
bar0088@argexpr2:~> export PATH=$PATH:/usr/sbin
bar0088@argexpr2:~> /sbin/ifconfig
bar0088@argexpr2:~> /opt/intel/mic/bin/micinfo
bar0088@argexpr2:~> /opt/intel/mic/bin/miccheck
[bar0088@cct407-6-mic0 bar0088]$ tail -n 25 /proc/cpuinfo
[bar0088@cct407-6-mic0 bar0088]$ head -5 /proc/meminfo
```

Ukázka 10: Zobrazení různých informací

5.2 Nativní režim

Jak již bylo zmíněno v předešlé kapitole, Xeon Phi disponuje dvěma základními režimy pro vykonávání aplikací [15], [16]. Prvním z nich je tzv. Nativní režim, kdy veškeré výpočty a simulace jsou prováděny přímo na koprocesoru, bez použití hostitelského CPU. V takovém případě i OpenMP nebo MPI hybridní kódy mohou být jednoduše zkompilovány a spuštěný přímo na koprocesoru. V takovém případě to můžeme chápat jako stroj s jednou sdílenou pamětí a 60 fyzickými jádry, kde každé dokáže zvládnout až 4 vlákna. Avšak sdílená paměť

karty (8 GB) je pro nativní režim limitující a nelze jej tak použít pro simulace žádají obrovskou paměť. V takovém případě je vhodné použít druhého režimu Offload.

Nyní si ukážeme jak spouštět aplikaci v nativním režimu [21]. Prvním krokem je mít připravený nějaký kód v jazyce C, C++ nebo Fortran, který chceme na kartě spouštět a je uzpůsoben pro běh v nativním režimu, tedy pokud využívá některých rozšíření jako OpenMP, MPI nebo MKL, musí to být v kódu definováno pomocí direktiv, aby koprocesor věděl, jak s danou částí kódu má zacházet. V našem případě se jedná o paralelní násobení matic s využitím OpenMP a MKL, a proto kód musí obsahovat direktivum `#pragma omp parallel`.

Po připojení k serveru pak přesuneme zdrojový soubor do našeho pracovního adresáře `/home/fei/bar0088/Workspace`, kde následně provedeme jeho kompilaci. V tuto chvíli se nám naskýtá možnost velkého množství nastavení kompilátoru. První je nutné nastavit samotný kompilátor s parametrem intel64, což provedeme následujícím příkazem `source /opt/intel/composer_xe_2013_sp1/bin/compilervars.sh intel64`. Dále si můžeme ověřit funkčnost správného nastavení příkazy `icc -V` (C kompiler), `icpc -V` (C++ kompiler) nebo `ifort -V` (Fortran kompiler, který v našem případě nebyl nainstalován). Dalším nutným parametrem pro kompilaci je volba jak má být soubor zkompilován. Nyní pro nativní režim použijeme parametr `-mmic` pro kompilaci pouze pro samotný koprocesor. Variantami může být například parametr `-xhost`, pro kompilaci pouze pro vykonání na hostitelském CPU nebo `-no-offload`, pro zakázání kompilace v offload režimu, případně povolení offload režimu příkazem `-offload-option`. Následně nastavíme parametr, kterým řekneme, jestli budou potřeba některé `rozšíření -openmp, -mpi, -mkl`. Nakonec zvolíme název našeho souboru `Matrix.c` a nastavíme název výstupního souboru `-o Matrix.out`. Celý příkaz pro kompilaci pak bude vypdat následovně viz Ukázka 11 [22].

```
bar0088@argexpr2:~/Workspace> source /opt/intel/composerxe/bin/compilervars.sh
intel64
bar0088@argexpr2:~/Workspace> icc -mmic -mkl -openmp -O3 -Wno-unknown-pragmas
-std=c99 -vec-report2 Matrix_main.c -o Matrix_native.out
Matrix_main.c(80): (col. 3) remark: loop was not vectorized: unsupported loop
structure
Matrix_main.c(65): (col. 7) remark: LOOP WAS VECTORIZED
Matrix_main.c(64): (col. 5) remark: loop was not vectorized: not inner loop
Matrix_main.c(75): (col. 3) remark: loop was not vectorized: loop was
transformed to memset or memcpy
remark: loop was not vectorized: operation cannot be vectorized
Matrix_main.c(75): (col. 3) remark: loop was not vectorized: not inner loop
```

```
Matrix_main.c(83): (col. 5) remark: loop was not vectorized: loop was
transformed to memset or memcpy
remark: loop was not vectorized: operation cannot be vectorized
Matrix_main.c(83): (col. 5) remark: LOOP WAS VECTORIZED
Matrix_main.c(83): (col. 5) remark: loop was not vectorized: not inner loop
Matrix_main.c(23): (col. 7) remark: loop was not vectorized: loop was
transformed to memset or memcpy
remark: loop was not vectorized: operation cannot be vectorized
Matrix_main.c(30): (col. 9) remark: LOOP WAS VECTORIZED
```

Ukázka 11: Nativní kompilace s údaji o vektorizaci

Nyní již máme náš kód zkompilovaný, ale před samotným spuštěním je ještě nutné jej nahrát z pracovního adresáře do koprocesoru, spolu s potřebnými knihovnami, jinak se nám spuštění nepodaří. V ukázce 12 je znázorněno jak soubory přesunout a také vidíme knihovnu `libiomp5.so`, kterou potřebujeme.

```
bar0088@argexpr2:~/Workspace> scp /opt/intel/composerxe/lib/mic/libiomp5.so
mic0:/tmp
libiomp5.so                                   100% 1116KB   1.1MB/s   00:00
bar0088@argexpr2:~/Workspace> scp Matrix_native.out mic0:/tmp
matrix.out                                    100%   28KB  27.7KB/s   00:00
```

Ukázka 12: Přesunutí souborů na koprocesor

Dalším krokem je již samotné spuštění výpočtu. Na koprocesor se připojíme přes již nakonfigurované zabezpečené připojení SSH příkazem `ssh mic0` a přejdeme do adresáře, kam jsme odeslali zkompilované soubory a knihovny, tedy `tmp`. V Ukázce 13 pak vidíme spouštěcí příkaz s parametrem řádu matice, počtu vláken a iterací. Dále je vhodné nastavit spouštěcí parametr `KMP_AFFINITY=balanced`, cestu ke sdíleným knihovnám a pro alokaci polí využití velkého stránkování například 100M. `OMP_NUM_THREADS` již nenastavujeme, protože počet vláken je již nastaven aplikací.

```
bar0088@argexpr2:~> ssh mic0
[bar0088@cct407-6-mic0 bar0088]$ cd /tmp
[bar0088@cct407-6-mic0 /tmp]$ ls
libimf.so    libintlc.so  libiomp5.so  libirng.so   libsvml.so
Matrix_native.out
[bar0088@cct407-6-mic0 /tmp]$ export LD_LIBRARY_PATH=/lib64
[bar0088@cct407-6-mic0 /tmp]$ export MIC_USE_2MB_BUFFERS=100M
[bar0088@cct407-6-mic0 /tmp]$ export KMP_AFFINITY=balanced
[bar0088@cct407-6-mic0 /tmp]$ ./Matrix_native.out 3000 240 10
Nasobeni matic s vyuzitim OpenMP a MKL
Vytvareni testovacich dat...
```

```
Zahajeni vypoctu...
./Matrix_native.out Pocet vlaken: 240 | Pocet iteraci: 10 |
Velikost matice: 3000 | Maximum: 0.693975 | Minimum: 0.425222 | Prumerne:
0.533791 | GFlop/s: 649.163
```

Ukázka 13: Spuštění aplikace v nativním režimu

Na výstupu dostaneme kromě informačních hlášení o použitém počtu vláken a zvoleném řádu matice také výsledný čas výpočtu. Pro objektivnost testu je výpočet proveden v `n` iteracích a výsledný čas je zobrazen jako minimální, maximální a průměrný. Tento čas jak si níže v jednotlivých testech ukážeme, se velmi odvíjí od zvoleného režimu a nastavených parametrů. V případě nativního režimu jsme omezeni do řádu matice asi 30 tisíc vzhledem k využití pouze 8 GB paměti koprocesoru. Právě tento nedostatek řeší offload režim, i když po výkonnostní stránce lehce ztrácí oproti nativnímu.

5.3 Offload režim

Druhým režimem pro vykonávání aplikací je tzv. Offload režim. Rozdíl oproti Nativnímu režimu je zejména v tom, že určité části kódu jsou vykonávány na koprocesoru a zbytek aplikace běží na hostitelském procesoru. K tomu slouží velké množství OpenMP pragmat (viz Tabulka 2), které leze jednoduše přidat do C/C++ nebo Fortran kódu, pro přesměrování vykonávání daného bloku na koprocesor. Tento přístup je velmi podobný pragma akcelerátoru zavedeným PGI kompilátorem nebo OpenACC k přesměrování kódu na GPGPU. Když Intel kompilátor narazí na offload pragma, vygeneruje kód pro oba, koprocesor i hosta. Kód pro přenos dat do koprocesoru je automaticky vytvořen kompilátorem, ale programátor jej může ovlivnit přidáním klauzule pro nastavení přenosu dat do offload pragma.

Název	**C/C++ syntaxe**	**Význam**
Offload pragma	`#pragma offload <klauzule> <funkce>`	Povolí vykonání další funkce na koprocesoru nebo hostitelském CPU
Offload transfer	`#pragma offload_transfer <klauzule>`	Inicializuje asynchronní datový přenos nebo zahájí synchronní datový přenos
Offload wait	`#pragma offload_wait <klauzule>`	Definuje pauzu pro dříve započaté asynchronní aktivity

Klíčové slovo pro definici proměnných a funkcí	`__attribute__((target(mic)))`	Definice pro kompilaci funkce nebo alokaci proměnné pro koprocesor i hostitelské CPU

Blok kódu	`#pragma offload_attribute(push, target(mic))`
	…
	`#pragma offload_attribute(pop)`
		Označení velkého bloku kódu pro koprocesor i hostitelské CPU

Specifikace cíle	`#pragma offload target(mic)`	Kde bude spouštěn kód
Vstup	`in(proměnné nebo pole)`	Zkopíruje data z hostitele na koprocesor
Výstup	`out(proměnné nebo pole)`	Zkopíruje data z koprocesoru na hostitele
Vstup i Výstup	`inout(proměnné nebo pole)`	Zkopíruje data z hostilele na koprocesor a zpět po dokončení offloadu
Nekopírovat žádná data	`nocopy(proměnné nebo pole)`	Data jsou pouze lokální
Pointer alokace paměti	`alloc_if(podmínka)`	Alokace paměti k uchování dat s odkazem na pointer pokud je podmínka splněna
Pointer dealokace paměti	`free_if(podmínka)`	Uvolnění paměti použité pointerem pokud je splněna podmínka

Tabulka 2: Syntaxe a význam OpenMP pragmat

Nyní se vrátíme k ukázce násobení matic, kterou jsme testovali dříve v nativním režimu. Teď se pokusíme test provést v offload režimu, kde je celá aplikace spouštěna z hostitelského prostředí, pouze samotná funkce je přesměrována k výpočtu na koprocesor. S tím souvisí nutnost nastavení přenosu dat mezi hostitelem a koprocesorem parametry `in`, `out`, `inout`. V Ukázce 14 vidíme kompilaci zdrojového kódu pro offload režim již bez parametru `-mmic` a následný výpis o vektorizaci.

```
bar0088@argexpr2:~> source /opt/intel/composerxe/bin/compilervars.sh intel64
bar0088@argexpr2:~> cd /home/fei/bar0088/Workspace/
bar0088@argexpr2:~/Workspace> icc -mkl -openmp -O3 -Wno-unknown-pragmas -
std=c99 -vec-report2 Matrix_main.c -o Matrix_offload.out
```

```
Matrix_main.c(81): (col. 3) remark: loop was not vectorized: unsupported loop
structure
Matrix_main.c(67): (col. 7) remark: LOOP WAS VECTORIZED
Matrix_main.c(66): (col. 5) remark: loop was not vectorized: not inner loop
Matrix_main.c(23): (col. 7) remark: LOOP WAS VECTORIZED
Matrix_main.c(23): (col. 7) remark: loop was not vectorized: not inner loop
Matrix_main.c(22): (col. 5) remark: loop was not vectorized: not inner loop
Matrix_main.c(30): (col. 9) remark: LOOP WAS VECTORIZED
Matrix_main.c(29): (col. 7) remark: loop was not vectorized: not inner loop
Matrix_main.c(28): (col. 5) remark: loop was not vectorized: not inner loop
```

Ukázka 14: Offload kompilace s údaji o vektorizaci

Nyní nám již nic nebrání zkompilovanou aplikaci spustit. Stačí opět nastavit spouštěcí parametry `KMP_AFFINITY`, načíst skript pro připojení sdílených knihoven OpenMP a MKL a pro zobrazení informací o průběhu výpočtů zapnout funkci `OFFLOAD_REPORT=[0|1|2|3]`. Tam se dozvíme informace o transferu dat, využití pointerů, Scatter/gather funkcí a podobně. V Ukázce 15 můžeme vidět názornou ukázku průběhu výpočtu a zkrácený výpis offload informací. Výsledkem výpočtu je opět po 10 iteracích opět minimální, maximální, průměrný čas a výpočetní výkon za sekundu v GFLOPS.

```
bar0088@argexpr2:~/Workspace> export KMP_AFFINITY=scatter
bar0088@argexpr2:~/Workspace> export MIC_USE_2MB_BUFFERS=100M
bar0088@argexpr2:~/Workspace> source /opt/intel/composerxe/bin/compilervars.sh
intel64
bar0088@argexpr2:~/Workspace> export OFFLOAD_REPORT=3
bar0088@argexpr2:~/Workspace> ./Matrix_offload.out 3000 240 10
Nasobeni matic s vyuzitim OpenMP a MKL
Vytvareni testovacich dat...
Zahajeni vypoctu...
[Offload] [HOST]  [State]   Initialize logical card 0 = physical card 0
[Offload] [MIC 0] [File]            Matrix_main.c
[Offload] [MIC 0] [Line]            16
[Offload] [MIC 0] [Tag]             Tag 0
[Offload] [HOST]  [Tag 0] [State]   Start Offload
[Offload] [HOST]  [Tag 0] [State]   Initialize function
__offload_entry_Matrix_main_c_16Multiplicationicc2056399518qATO8W
[Offload] [HOST]  [Tag 0] [State]   Create buffer from Host memory
[Offload] [HOST]  [Tag 0] [State]   Create buffer from MIC memory
 [Offload] [HOST]  [Tag 0] [State]   Create buffer from MIC memory
[Offload] [HOST]  [Tag 0] [State]   Send pointer data
[Offload] [HOST]  [Tag 0] [State]   CPU->MIC pointer data 144000000
[Offload] [HOST]  [Tag 0] [State]   Gather copyin data
```

```
[Offload] [HOST]  [Tag 0] [State]   CPU->MIC copyin data 52
[Offload] [HOST]  [Tag 0] [State]   Compute task on MIC
[Offload] [HOST]  [Tag 0] [State]   Receive pointer data
[Offload] [HOST]  [Tag 0] [State]   MIC->CPU pointer data 72000000
[Offload] [MIC 0] [Tag 0] [State]   Start target function
__offload_entry_Matrix_main_c_16Multiplicationicc2056399518qATO8W
...
./Matrix_offload.out Pocet vlaken: 240 | Pocet iteraci: 10 | Velikost matice:
3000 | Maximum: 0.948362 | Minimum: 0.788593 | Prumerne: 0.900152 | GFlop/s:
459.939
```

Ukázka 15: Spuštění aplikace v offload režimu

Takto jednoduše lze aplikace spouštět v offload režimu, kdy se v případě velké kapacity paměti na hostiteli nemusíme spoléhat pouze na nativní režim. Tento režim však vyžaduje pro správné vykonání úpravu zdrojového kódu programátorem a přesné definice dle tabulky výše, bez které není možné správně provést přesměrování výpočtů na koprocesor. Právě tato komplikace odpadá v nativním režimu, kdy stačí mít pouze aplikaci využívající OpenMP pro paralelizaci na všechny jádra a znalost jak ji spustit. Dalším důležitým prvkem pro co nejvyšší výpočetní výkon je optimalizace kódu, kdy je kladen důraz na paralelizaci a práci s pamětí. Neefektivní přidělování paměti a zbytečné transfery dat rapidně ubírají na výsledném výkonu. Stejně tak využití jen malého počtu jader, kdy jsou výsledky ne o moc lepší než v případě hostitelského Intel Xeonu jak si ukážeme níže.

5.4 Host-only režim

Posledním z možných typů spouštění aplikací je tzv. Host-only režim, nebo také hostitelský režim, kdy aplikaci kompilujeme pouze pro vykonání na hostiteli, tedy na procesoru Intel Xeon E5 4610. Tento procesor disponuje 6 jádry a je schopen vykonávat paralelní aplikace ve 12 vláknech. V tomto režimu postupujeme podobně jako u předešlých dvou, jen při kompilaci použijeme parametry `-xhost` a `-no-offload`. Před spuštěním pak vypneme použití koprocesoru příkazem `export MKL_MIC_ENABLE=0`, kdy nedojde ani k automatickému offloadu. Koprocesor v takovém případě nebude vůbec využit. Tento režim není vhodný pro vysoce paralelní výpočty, kde je koprocesor mnohem výkonnější, ale spíše pro aplikace využívající méně vláken nebo pro porovnání výpočetního výkonu koprocesoru a hosta [17].

5.5 Volitelné parametry kompilace

Při kompilaci máme možnost nastavit velké množství parametrů, které jsou buď funkčního, nebo informativního charakteru [18]. Mezi ty funkční patří parametr pro nastavení optimalizace kompileru, který provedeme příkazem `-O[0|1|2|3]`, přičemž defaultní nastavení je `-O2`. V tomto případě číslo znamená, jak velký důraz je kladen při kompilaci na výsledný výkon. V případě nastavení `-O0` je kladen důraz na rychlost kompilace, zatím co u `-O3` je kladen důraz na výkon. Jedná se o nejagresivnější optimalizaci zaměřenou na nejvyšší výpočetní výkon. Dalším je nastavení názvu a formátu výstupního souboru příkazem `-o vystupni_soubor.out`.

Dalším volitelným parametrem je tzv. report o vektorizaci, který nám říká, které části kódu byly při kompilaci vektorizovány a které ne. Kompilátor detekuje operace v programu, které mohou být provedeny paralelně a převádí sekvenční operace na paralelní. Například převádí sekvenční SIMD instrukce, které zpracovávají 2, 4, 8 nebo až 16 prvků do paralelního provozu v závislosti na typu dat. Použitím parametru -vec umožňuje vektorizaci na výchozí úroveň optimalizace pro hostitelský CPU nebo koprocesor Xeon Phi. K samotnému výpisu pak slouží příkaz `-vec-report[0|1|2|3|4|5]`, kde číslo označuje množství zobrazených detailů.

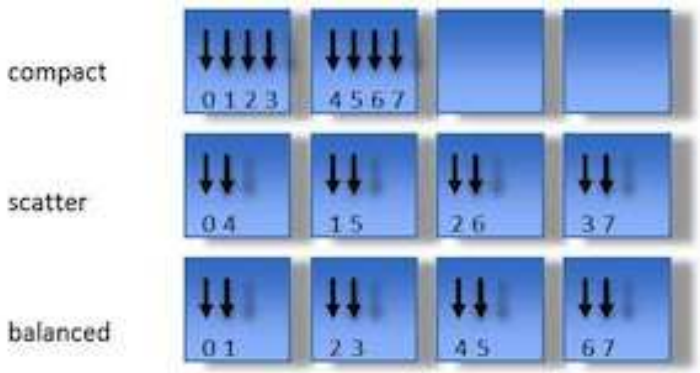
Při kompilaci OpenMP aplikací je nutné uvést parametr `-openmp`. Tím se nám zpřístupní možnost vypsání tzv. diagnostických zpráv pomocí příkazu `-openmp_report[0|1|2]`.

Mezi velmi často používané volitelné parametry kompilace patří parametr `-Wno-unknown-pragmas`, který detekuje nerozeznané nebo chybné pragmata a informuje nás o tom výpisem. V neposlední řadě je vhodné také zmínit parametr `-std=c99` pro specifikaci použitého programovacího jazyku kompileru. To je potřeba například kvůli způsobu deklarace polí.

5.6 Vstupní spouštěcí parametry

Před samotným spuštěním máme možnost volby spouštěcích parametrů (Environment variables) [14]. Tyto parametry jsou nepovinné a koprocesor je má nastaveny na defaultní hodnoty. Pokud je však nastavíme, můžeme razantně zlepšit výpočetní výkon u některých aplikací. Na začátku je nutné nastavit parametr `MIC_ENV_PREFIX=MIC`, kterým řekneme, že chceme použít vlastní nastavení proměnných a zkopírovat je na koprocesor. Mezi další patří například parametr pro nastavení maximálního počtu vláken `OMP_NUM_THREADS`. Ten je vhodné zkoušet v rozmezí 60 až 240. V případě použití více koprocesorů můžeme přesně definovat kolik vláken má který koprocesor využít příkazem `MIC_1_OMP_NUM_THREADS`. Velmi důležitým

parametrem je `KMP_AFFINITY`, kterým řídíme rozdělování vláken pro každé jádro nastavením hodnot `compact, scatter, balanced` viz Obrázek 18. Obecně nejlepších výkonů dosahujeme s použitým nastavením `balanced`. Pokud chceme využívat automatický offload režim při spouštění aplikace, jen nutné nastavit parametr `MKL_MIC_ENABLE=1` a ověřit načtení mkl a mic knihoven příkazem `MODULE LOAD MKL MIC`. Pro účely ladění je doporučeno mít zapnutý tzv. offload report nastavený příkazem `OFFLOAD_REPORT=[0|1|2|3]`, kterým získáme informace o vykonávání výpočtů a případně o výskytu chyb.



Obrázek 18: Rozdělení KMP_AFFINITY [1]

6 Testování

Nyní, když jsme si vysvětlili všechny režimy, popsali a ukázali všechny důležité funkce pro kompilaci a spouštění aplikací na koprocesoru Xeon Phi, můžeme přejít do fáze testování naší aplikace násobení matic. Až nyní zjistíme, jakého výkonu můžeme při spuštění aplikace dostáhnout, v jakém režimu je nejvhodnější spouštět, při jakém počtu vláken a jak velkou roli hrají různé detaily popsané výše. Pro programátora je velmi důležité vědět, jaké parametry zvolit pro danou aplikaci, aby dosáhl nejlepšího možného výkonu. Z následujícího příkladu vyplývá, že výsledný rozdíl při nastavení jednoho spouštěcího parametru prostředí není nijak extrémně velký, avšak pokud jich nastavíme několik, dosáhneme v celkovém výsledku poměrně slušného zlepšení výkonu [23].

První testovanou ukázkou je násobení dvou čtvercových matic řádu 3000 ve všech 3 režimech – nativně, offload, host-only. Měřenou veličinou je zde výsledný čas, který je vytvořený průměrem ze všech testovaných iterací kvůli co nejvyšší objektivitě. Druhou veličinou je pak výpočetní výkon GFLOPs, tedy počet operací v plovoucí řádové čárce za sekundu. Modifikovanou veličinou je pak počet vláken, který postupně narůstá od 4 do 240, kdy 12 vláken je maximum hostitelského 6 jádrového procesoru Intel Xeon E5 4610 a 240 vláken 60 jádrového koprocesoru Intel Xeon Phi 5110P.

Režim / Počet vláken	Nativní režim	Offload režim	Host-only režim
4	5,859s / 55 GFLOPs	0,880s / 368 GFLOPs	1,223s / 34 GFLOPs
12	1,952s / 165 GFLOPs	0,873s / 371 GFLOPs	0,616s / 88 GFLOPs
40	0,726s / 442 GFLOPs	0,867s / 373 GFLOPs	-
120	0,336s / 715 GFLOPs	0,873s / 370 GFLOPs	-
150	0,281s / 790 GFLOPs	0,853s / 379 GFLOPs	-
180	0,426s / 552 GFLOPs	0,880 / 367 GFLOPs	-
240	0,463s / 547 GFLOPs	0,882 / 352 GFLOPs	-

Tabulka 3: Násobení matic řádu 3 000 s nastaveným KMP_AFFINITY=balanced

V Tabulce 3 je možné vidět, jak tento test dopadl. V nativním režimu můžeme pozorovat postupný nárůst výkonu se zvyšujícím se počtem vláken, kde kolem 150 vláken zaznamenáváme nejvyšší výkon, který pak mírně klesá. V offload režimu však změna počtu vláken nehraje téměř žádnou roli a výsledný výkon je téměř konstantní. Zajímavé je však srovnání výkonu při 12 vláknech v host-only režimu, kde je jasně vidět převaha procesoru Intel

Xeon, který je se svými 6 jádry výkonnější. Je tomu díky frekvenci, kdy procesor má taktované jádra na 2,4 GHz (2,9 GHz Turbo), zatímco koprocesor pouze 1,053 GHz. Nejvyšší výkon v tomto testu pak byl dosažen v nativním režimu při použití 156 vláken, kdy byl výpočet proveden v průměru za 0,259s při výkonu 802 GFLOPs.

Výpočet operací v plovoucí řádové čárce za sekundu v jednotkách GFLOPs provádíme spuštěním aplikace s velkými vstupními daty a měříme výpočetní čas. Algoritmus násobení matic (Ukázka 16) provádí 2 operace v plovoucí řádové čárce pro každou smyčku, které když vynásobíme počtem smyček, dostaneme celkový počet operací. Výsledný výpočetní výkon získáme vydělením počtu operací a výpočetního času, tím dostaneme výsledek v jednotkách FLOPs a vynásobením 10^{-9} se dostaneme do řádu GFLOPs.

$$Počet\ operací = 2 \times Řád\ matice^3$$

$$GFLOPs = 10^{-9} \times \left(\frac{Počet\ operací}{Výpočetní\ čas}\right)$$

Následující test, jehož výsledek ukazuje Tabulka 4, byl proveden s vyššími vstupními daty než první test. Počet iterací výpočtu byl opět 10, ale řád matice narostl z původních 3 000 na 10 000 a parametr `KMP_AFFINITY` byl v tomto případě `scatter`, oproti předchozímu `balanced`. Z výsledků vyplývá, že nárůst či naopak pokles výkonu je v nativním režimu podobný jako v předcházejícím testu, ale rozdíl nastal v počtu vláken, kdy bylo dosaženo nejvyššího výkonu a to 118 při 792 GFLOPs. U offload režimu došlo k mírnému nárůstu výkonu, ale opět vidíme téměř konstantní výsledky jako v předchozím případě, kdy změna počtu vláken téměř nehraje roli. V případě využití 4 nebo 12 vláken opět vítězí samotný Intel Xeon bez použití koprocesoru, kdy rozdíl při 4 vláknech v porovnání s nativním režimem je téměř dvojnásobný a při 12 vláknech dokonce trojnásobný. Z tohoto pohledu je jasně vidět nevhodnost použití koprocesoru pro jakékoliv aplikace jedno vlákno nebo řádově jednotky vláken. Samotný počet vláken, při kterém bude aplikace dosahovat nejvyššího výkonu nelze předem určit. Vždy se neobejdeme bez důsledného testování, kdy hledáme optimální počet vláken.

Režim / Počet vláken	Nativní režim	Offload režim	Host-only režim
4	116,837s / 53 GFLOPs	26,145s / 421 GFLOPs	61,540s / 32 GFLOPs
12	73,952s / 163 GFLOPs	26,052s / 423 GFLOPs	21,590s / 92 GFLOPs
40	23,423s / 482 GFLOPs	25,986s / 426 GFLOPs	-
120	13,927 / 787 GFLOPs	25,884s / 428 GFLOPs	-
150	15,753s / 643 GFLOPs	25,927s / 425 GFLOPs	-
180	19,929s / 584 GFLOPs	25,981 / 426 GFLOPs	-
240	24,347s / 492 GFLOPs	25,896 / 428 GFLOPs	-

Tabulka 4: Násobení matic řádu 10 000 s nastaveným KMP_AFFINITY=scatter

7 Závěr

Na počátku mě toto téma zaujalo již při volbě diplomové práce. Na internetu jsme se mohli dočíst o novince on Intelu postavené na koncepci Larrabe, která bude dosahovat závratného výkonu a doslova smete všechnu konkurenci. Mít možnost pracovat s takovou technologií byla pro mě výzva, protože to byla příležitost získat v tomto směru nové neocenitelné znalosti. Na počátku jsem si ani neuvědomoval, kolik problémů s tím spojených mě později čeká. Bylo velmi složité nalézt nějaké věrohodné a podrobnější informace, jelikož se jednalo o naprostou novinku, která se teprve chystá na trh. Později, když se tyto informace začaly postupně objevovat, byly mnohé nepřesné nebo nesmyslné. Tak vznikala první část této práce zabývající se architekturou. Druhá část byla o poznání lepší, jelikož technologie OpenCL a CUDA jsou tu již mnoho let a jejich vlastnosti jsou obecně známy. To se však nedalo říci o vlastnostech a možnostech použití Intel Xeonu Phi. Netušil jsem, že nejobtížnější část mě teprve čeká a bude na ni tak málo času. Možnost pracovat na koprocesoru, tedy vůbec první vzdálené připojení k serveru bylo jen necelé 2 měsíce před odevzdáním této práce. Času bylo málo a práce mnoho, proto jsem se této práci věnoval téměř denně a pomalu začínal chápat jak vše funguje. Vyskytly se i nějaké problémy, které jsem se snažil ihned řešit. Za zmínku stojí například konfigurace vzdáleného připojení, načtení sdílených knihoven nebo nastavení kompileru, kdy jsem problém řešil pomalu týden přímo s Intel supportem. Podařilo se mi spustit první aplikaci v nativním režimu, později si napsat vlastní, upravit ji pro spouštění v offload režimu a provést výkonnostní testy koprocesoru.

Rozsah tématu je tak veliký, že by vydal na nejednu knihu a člověk by jeho studiem mohl strávit mnoho let. Tato diplomová práce je zaměřena na samotný základ práce s koprocesorem, aby čtenáři poskytla obecný přehled o možnostech využití v praxi a porovnání s ostatními technologiemi včetně konkrétních funkčních ukázek. V budoucnu pak může sloužit jako výchozí bod pro ty, kteří by se tímto tématem rádi zabývali a chtěli všechny základní informace pohromadě včetně rad, čeho se při práci s koprocesorem Intel Xeon Phi zpočátku vyvarovat.

8 Reference

[1] Jim Jeffers, James Reinders: *Intel Xeon Phi Coprocessor Hight-Performance Programming.*
Elsevier Inc. 2013, 409 s. ISBN: 978-0-12-410414-3

[2] Rezaur Rahman: *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers.*
Apress Media,LLC. 2013, 103 s. ISBN: 978-1-4302-5927-5

[3] Andrey Vladimirov, Vadim Karpusenko : *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors.*
Colfax International. 2013, 517 s. ISBN: 978-0-9885234-1-8

[4] INTEL [online]. 2013 [cit. 2014-02-18] *Intel Xeon Phi – Intel Developer Zone.*
URL: < http://software.intel.com/en-us/mic-developer/ >

[5] INTEL [online]. 2013 [cit. 2014-01-07] *Intel MIC User Forum.*
URL: < http://software.intel.com/en-us/mic-developer />

[6] INTEL [online]. 2013 [cit. 2013-11-16] *Intel Xeon Phi Coprocessor – System Software Developers Guide.*
URL: <http://download-software.intel.com/sites/default/files/managed/b5/83/intel-xeon-phi-systemssoftwaredevelopersguide.pdf/>

[7] INTEL [online]. 2012 [cit. 2013-03-27] *Intel Xeon Phi Coprocessor - The Architecture.*
URL: <http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner/>

[8] INTEL [online]. 2012 [cit. 2013-03-27] *Intel Xeon Phi Coprocessor – Developer's Quick Strat Guide.*
URL: <http://software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf/>

[9] KHRONOS [online]. 2009 [cit. 2013-11-20] *The OpenCL Specification.*
URL: <http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf/>

[10] AMD [online]. 2013 [cit. 2013-11-25] *Programming in OpenCL.*
URL: <http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/>

[11] NVIDIA [online]. 2013 [cit. 2013-11-26] *CUDA Toolkit Documentation.*
URL: <http://docs.nvidia.com/cuda/index.html/>

[12] NVIDIA [online]. 2013 [cit. 2013-11-26] *CUDA C Programming Guide.*
URL: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>

[13] PRACE [online]. 2013 [cit. 2014-01-07] *Intel Xeon Phi – Best Practice Guide.*
URL: < http://www.prace-ri.eu/Best-Practice-Guide-Intel-Xeon-Phi-HTML?lang=en/>

[14] INTEL [online]. 2013 [cit. 2014-03-22] *Intel Xeon Phi –Programming Environment.* URL: < http://software.intel.com/en-us/articles/intel-xeon-phi-programming-environment/>

[15] VirginiaTech [online]. 2013 [cit. 2014-03-22]*Using the Intel MIC cards on BlueRidge.* URL: < http://www.arc.vt.edu/resources/hpc/blueridge_mic.php/>

[16] IT4Innovations [online]. 2013 [cit. 2014-03-23] *A guide to Intel Xeon Phi usage.* URL: < https://support.it4i.cz/docs/anselm-cluster-documentation/software/intel-xeon-phi/>

[17] JLAB [online]. 2013 [cit. 2014-03-23] *Intel Xeon Phi MIC Cluster.* URL: < https://wiki.jlab.org/cc/external/wiki/index.php/Intel_Xeon_Phi_(MIC)_Cluster>

[18] HPC Calcul Québec [online]. 2013 [cit. 2014-03-23] *Intel Xeon Phi – Workshop* URL: < http://www.hpc.mcgill.ca/downloads/xeon_phi_workshop_nov2013 /phiwksp.pdf/>

[19] SCAI [online]. 2013 [cit. 2014-03-023] *MIC Tutorial.* URL: < http://www.hpc.cineca.it/content/mic-tutorial/>

[20] TACC [online]. 2013 [cit. 2014-03-23] *Intel Xeon Phi – Stampede User Guide.* URL: < https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide#overview/>

[21] INTEL [online]. 2013 [cit. 2014-03-22] *Building Native Applications for Intel Xeon Phi.* URL: < http://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors/>

[22] INTEL [online]. 2013 [cit. 2014-03-023] *Intel Compiler 13.1 User Guide.* URL: < http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/ composerxe/compiler/cpp-lin/index.htm/>

[23] INTEL [online]. 2013 [cit. 2014-03-023] *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors.* URL: < http://download-software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf/>

A Zdrojové kódy

- Matrix_main.c - zdrojový kód aplikace násobení matic pro kompilaci
- Matrix_native.out - zkompilovaná aplikace pro nativní režim
- Matrix_offload.out - zkompilovaná aplikace pro offload režim
- Matrix_host.out - zkompilovaná aplikace pro hostitelský režim